Data Structures

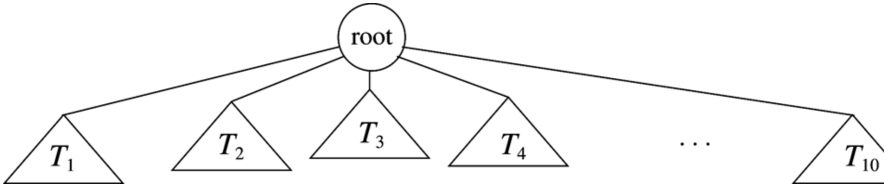
Trees

Hikmat Farhat

June 27, 2018

Introduction

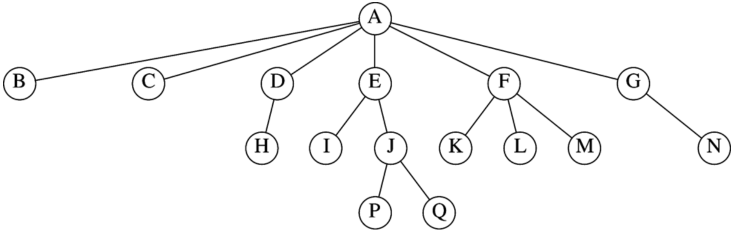
- A **tree** is a collection of nodes defined recursively as
  - A **root** node connected to zero or more subtrees by an **edge**



- If R is the **root** of the tree then the root of each subtree is called the **child** of R.
- R is called the **parent** of the root of each subtree.
- Using the recursive definition we can deduce that a tree with N nodes has $N - 1$ edges since each node, expect the root, has exactly one parent.
- Each node of the tree can have 0 or more children.
- A node that has 0 children is called a **leaf**
- Nodes that have the same parent are called **siblings**

- A sequence of nodes $n_1, \ldots n_k$ with n_i being the parent of n_{i+1} is called a **path** from n_1 to n_k.
- The **length** of a path is the number of edges on the path.
- For any node n_i the **depth** of n_i is the length of the unique path from the root to n_i.
- The **height** of a node n_i is the length of the longest path from n_i to a leaf.

Example



- Node E has depth 1 and height 2
- The parent of E is A.
- Nodes B,C,D, E, F and G are siblings.
- Note that the depth of the root, A is 0.

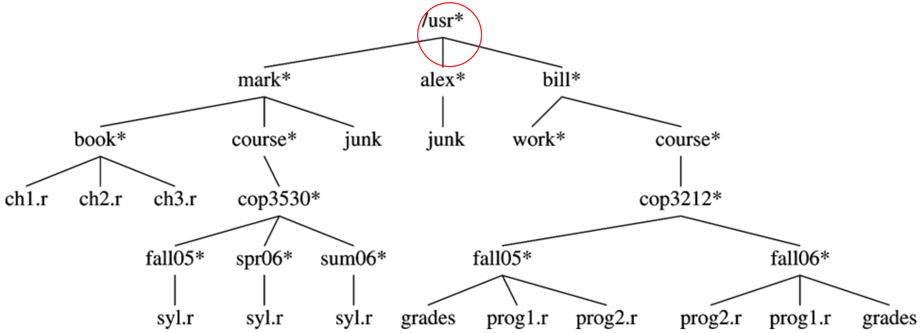
Traversal of Trees

- The nodes of a tree can be processed in three ways
  1. Preorder: each node is processed **before** its children.
  2. Postorder: each node is processed **after** its children.
  3. Inorder (in the case of a binary tree): left child is processed first, then the node then the right child.

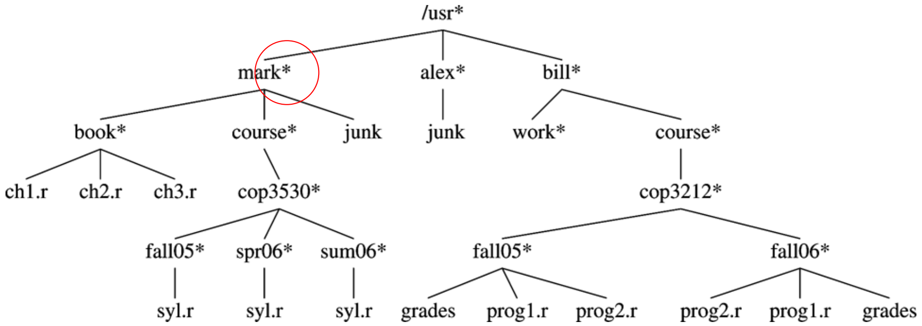
Preorder Traversal

Output

usr



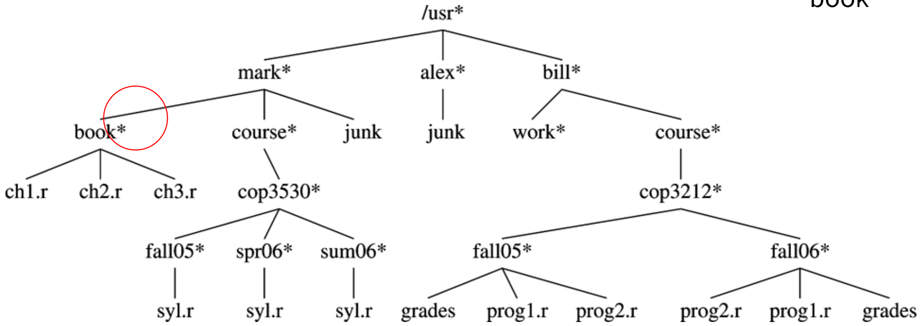
Preorder Traversal



Output

usr
mark

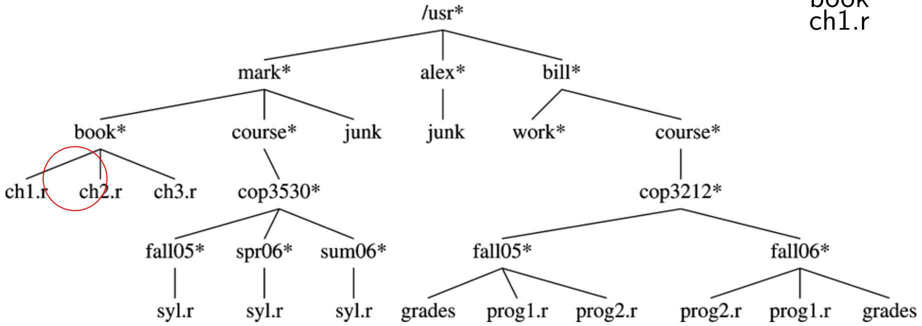
Preorder Traversal



Output

usr
mark
book

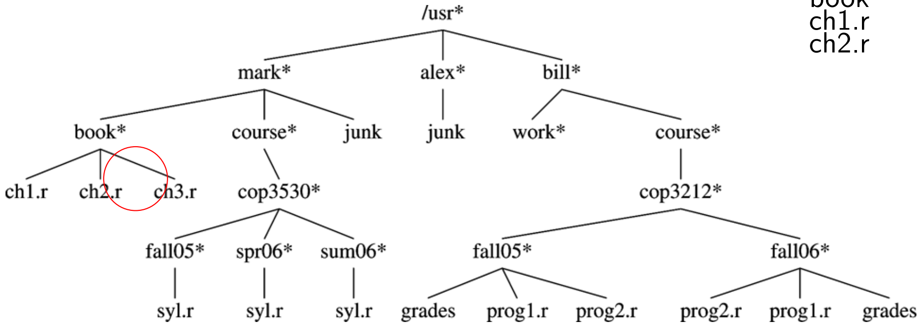
Preorder Traversal



Output

usr
mark
book
ch1.r

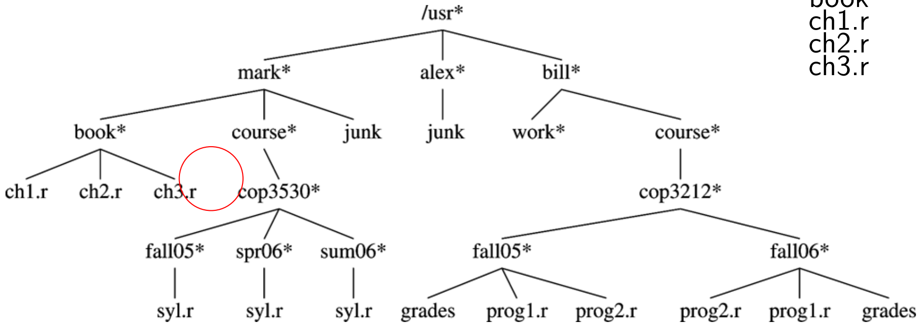
Preorder Traversal



Output

usr
mark
book
ch1.r
ch2.r

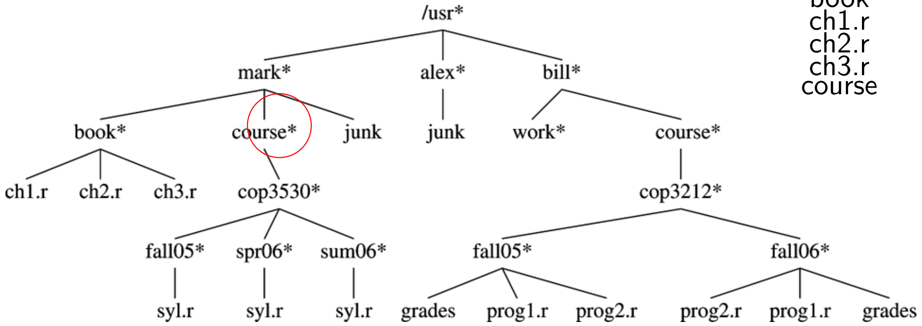
Preorder Traversal



Output

usr
mark
book
ch1.r
ch2.r
ch3.r

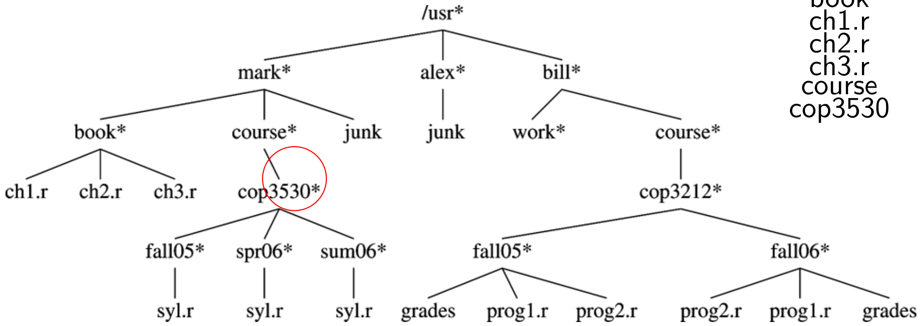
Preorder Traversal



Output

usr
mark
book
ch1.r
ch2.r
ch3.r
course

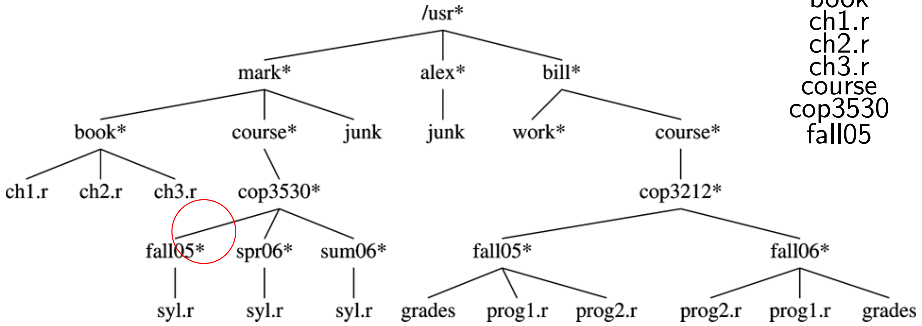
Preorder Traversal



Output

usr
mark
book
ch1.r
ch2.r
ch3.r
course
cop3530

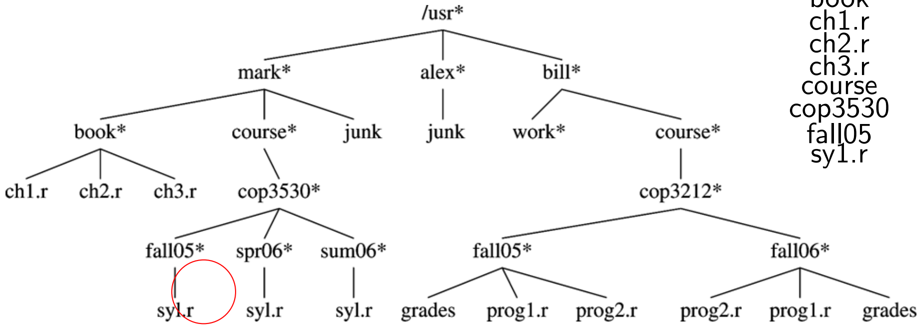
Preorder Traversal



Output

usr
mark
book
ch1.r
ch2.r
ch3.r
course
cop3530
fall05

Preorder Traversal

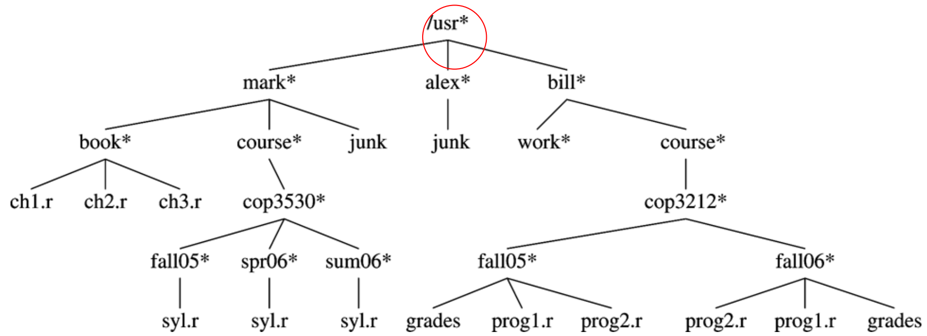


Output

usr
mark
book
ch1.r
ch2.r
ch3.r
course
cop3530
fall05
sy1.r

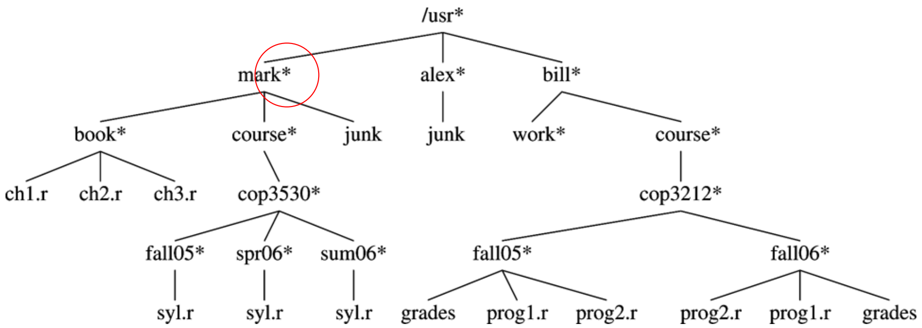
Postorder Traversal

Output



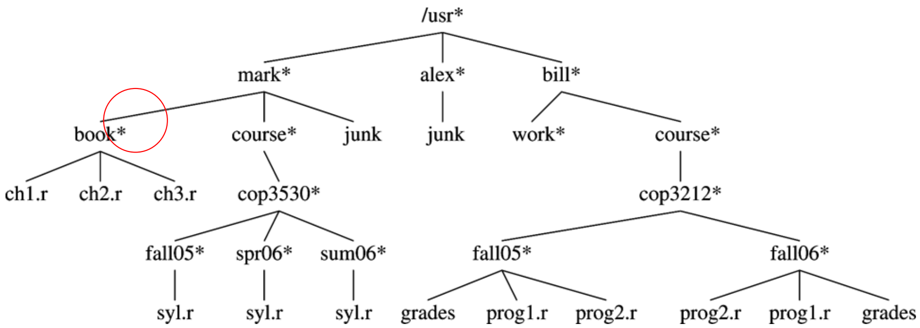
Postorder Traversal

Output



Postorder Traversal

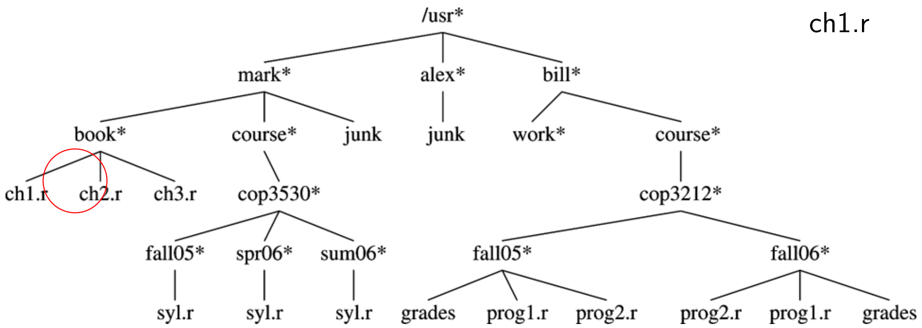
Output



Postorder Traversal

Output

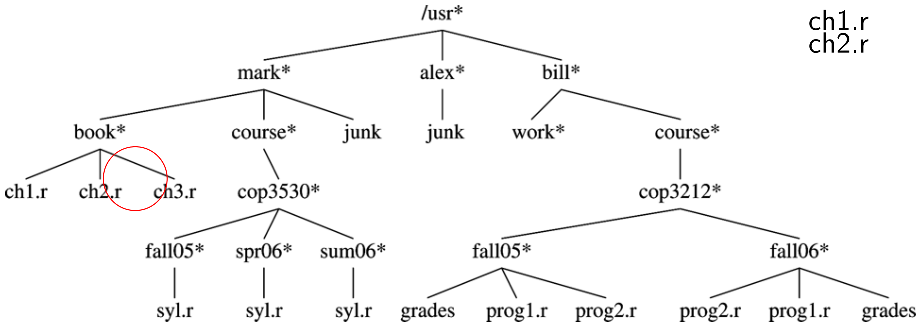
ch1.r



Postorder Traversal

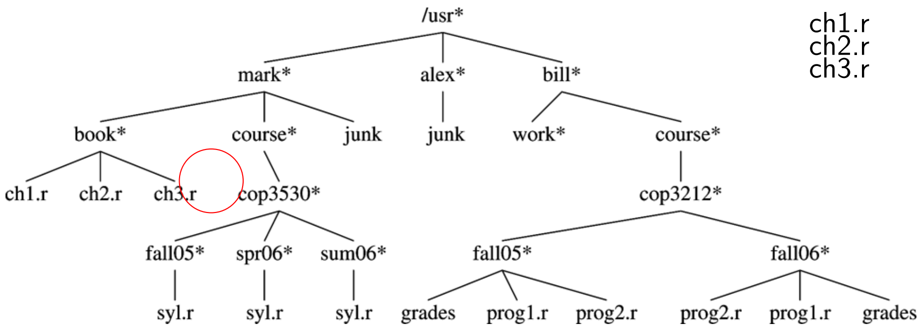
Output

ch1.r
ch2.r



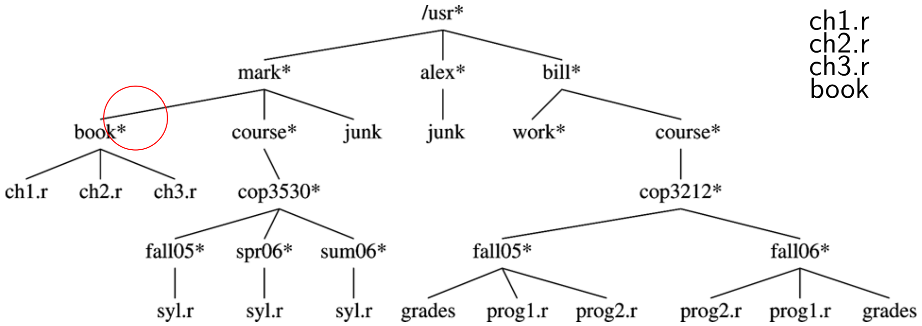
Postorder Traversal

Output



ch1.r
ch2.r
ch3.r

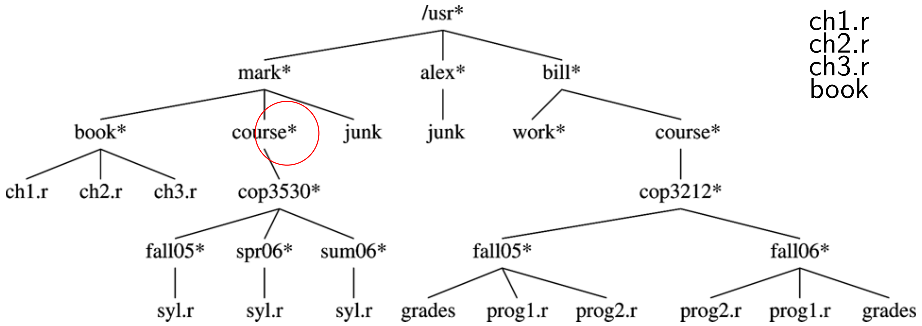
Postorder Traversal



Output

ch1.r
ch2.r
ch3.r
book

Postorder Traversal

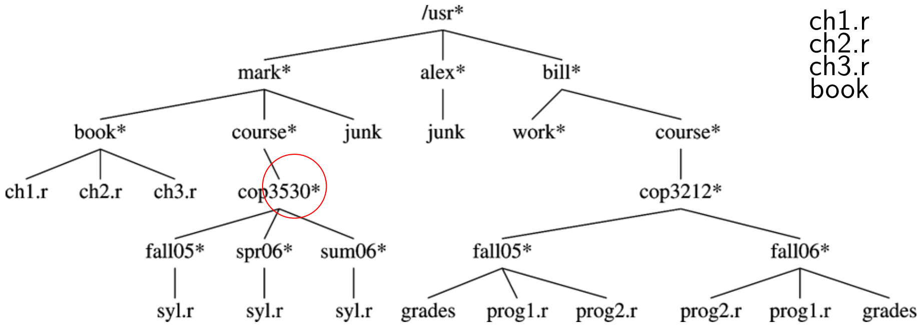


Output

ch1.r
ch2.r
ch3.r
book

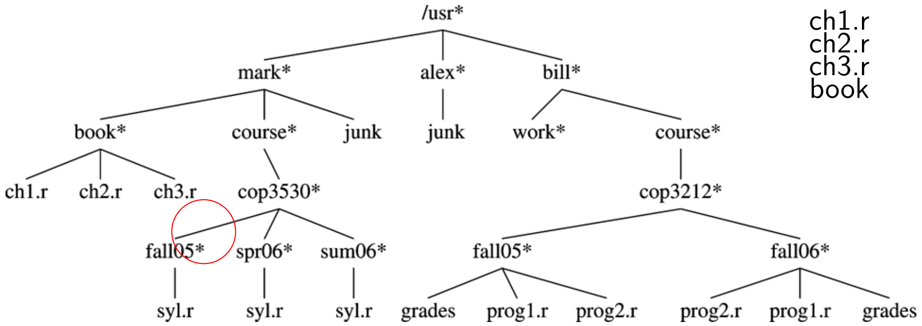
Postorder Traversal

Output



ch1.r
ch2.r
ch3.r
book

Postorder Traversal

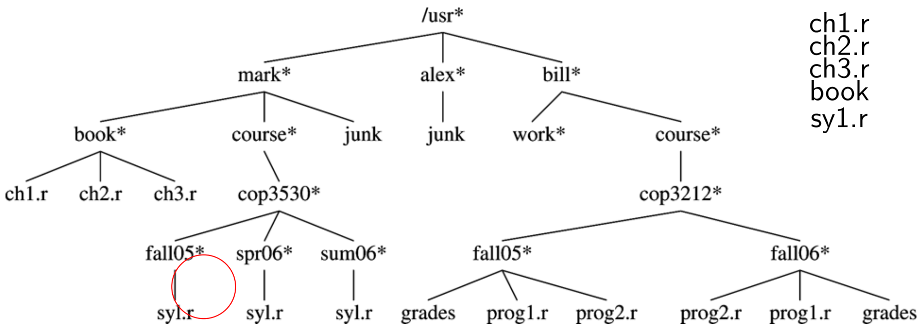


Output

ch1.r
ch2.r
ch3.r
book

Postorder Traversal

Output



ch1.r
ch2.r
ch3.r
book
sy1.r

Recursive Implementation

```
void postorder(Node *t){
    if(t==NULL)return;
    postorder(t->left);
    postorder(t->right);
    cout<<t->data<<endl;

}
void preorder(Node *t){
    if(t==NULL)return;
    cout<<t->data<<endl;
    preorder(t->left);
    preorder(t->right);


}
```

Binary Trees

- A binary tree is a tree in which **no** node can have more than **two children**.
- The **height** of a tree is equal to the height of its root.
- The **depth** of a tree is equal to the depth of its deepest leaf which is equal to the height of the tree.
- The depth of an average (random) binary tree of N nodes is $O(\sqrt{N})$
- A **complete** binary tree is a tree in which all levels are full with the possible exception of the last level where the nodes are to the left.
- A **full** binary tree is a binary tree in which each node is either a leaf or has exactly two children.
- A **Perfect** binary tree is a tree that is full and all leaves have the same depth.

Inorder Traversal of BT

```
void inorder(Node *t){
   if(t==NULL)return;
   inorder(t->left);
   cout<<t->data<<endl;
   inorder(t->right);
}
```

Binary Search Tree

- A binary search tree is a binary tree with a special property
- for every node X the values of the nodes to the left of X are smaller than the value of X and the ones to the right of X are larger than the value of X.
- the average height of a binary search tree with N nodes is $O(\log N)$.

Example

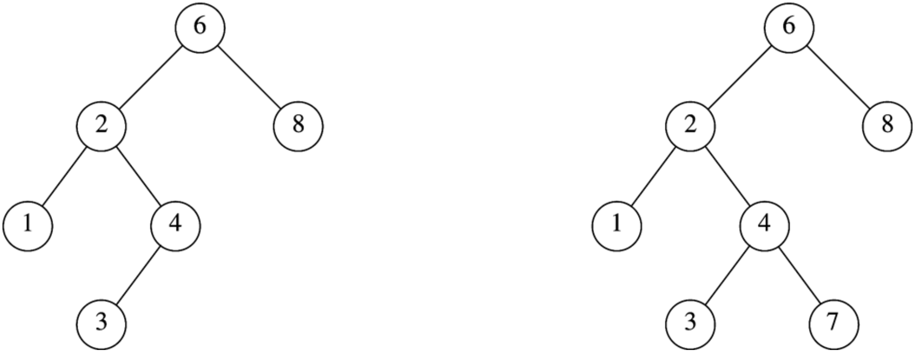


Figure : Left tree is a BST the right is not

- Why are BST important?
- Because it is fast to find an element in the tree.
- if we are looking for value x and reach node y
  - If $x > y$ we go right
  - If $x < y$ then we proceed to the left
- Similarly it is fast to find the minimum and maximum of a collection.

BST interface

```
BST(const Comparable &x){
    root=new Node(x,NULL,NULL);
}
int numNodes();
int numLeaves();
void insert(const Comparable & x);
void remove(const Comparable & x);
Comparable findMin();
Comparable findMax();
int height();
void print();
void printPreorder(){printPreorder(root);}
void printPostorder(){printPostorder(root);}
void printInorder(){printInorder(root);}
```

BST interface

```
    private:
    struct Node {
        Node *left;
        Node *right;
        Comparable data;
        Node(const Comparable &x,Node *l,Node *r)
        :data(x),left(l),right(r){}
   };
   Node *root;
   void print(Node *n,int level);
   void insert(const Comparable &x,Node * &t);
   void remove(const Comparable &x, Node * & t);
   void printPreorder(Node *n);
   void printPostorder(Node *n);
   void printInorder(Node *n);
```

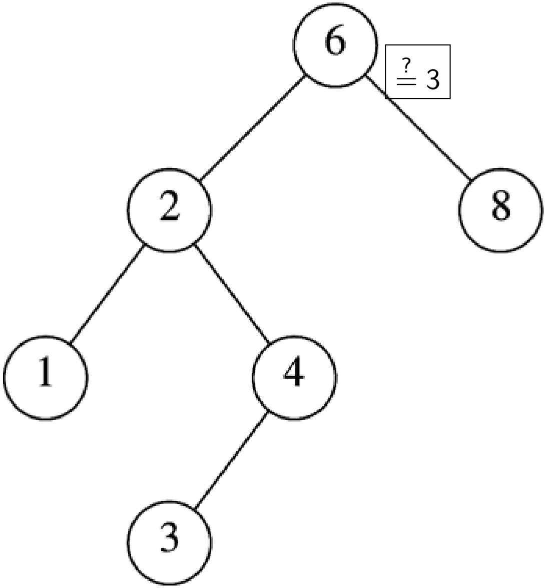
BST interface

```
    Node * findMin(Node *t);
    Node * findMax(Node *t);
    int height(Node *t);
    int numNodes(Node *t);
    int numLeaves(Node *t);
```

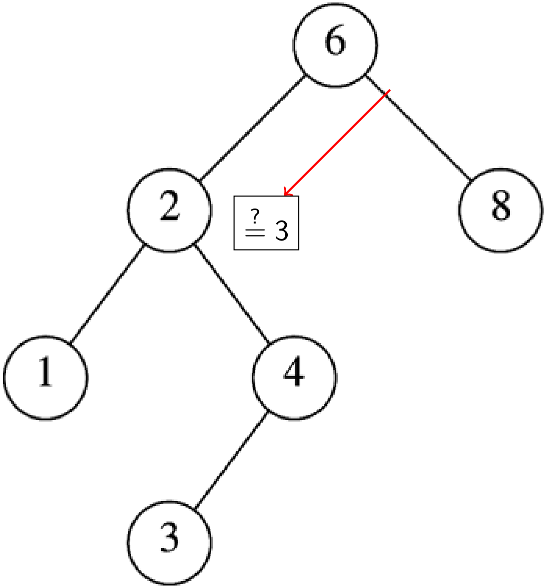
Finding a Value in a BST

- To find a value compare the current note with that value:
  - If the node is null return false, otherwise
  - If the value is equal to the node's value return true
  - If the value is less than the node's value then check the left child
  - If the value is more than the node's value then check the right child

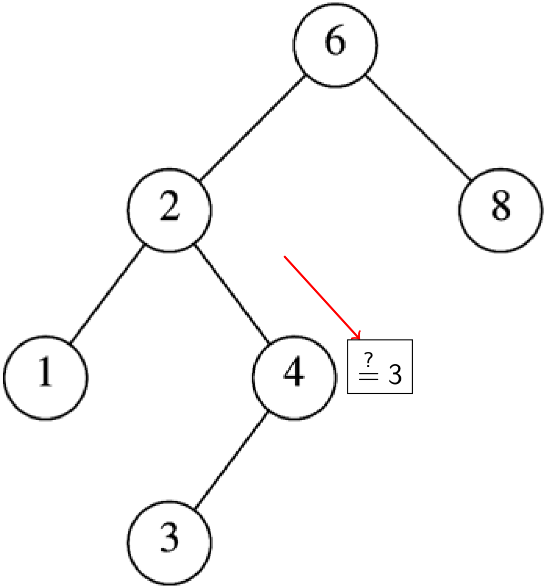
Example Search



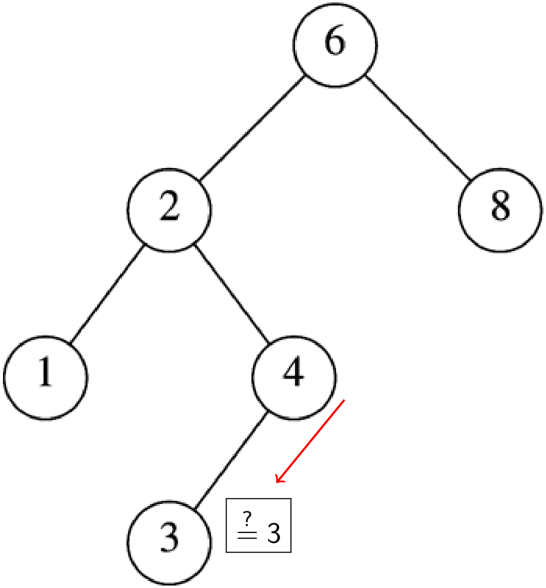
Example Search



Example Search



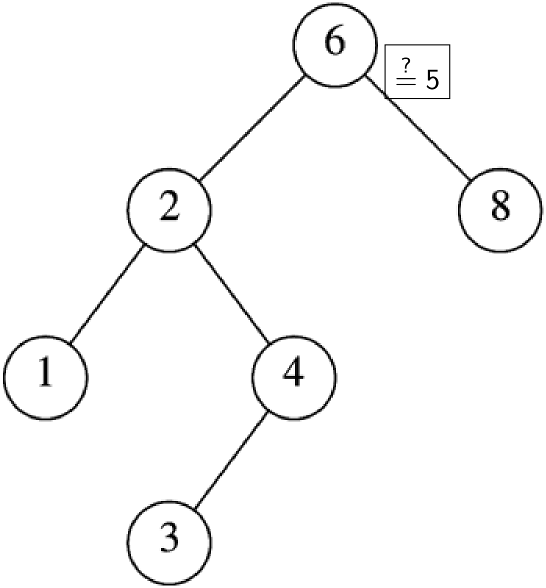
Example Search



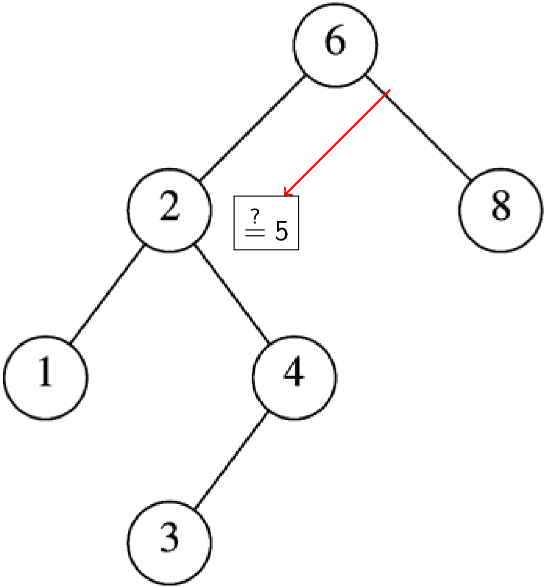
Inserting an Element

- Inserting an element is similar to "finding" an element.
  - if the value to be inserted is less than the current node go **left**
  - If the value to be inserted is more than the current node go **right**
  - Keep doing this until the node to be compared does not exist then create one.

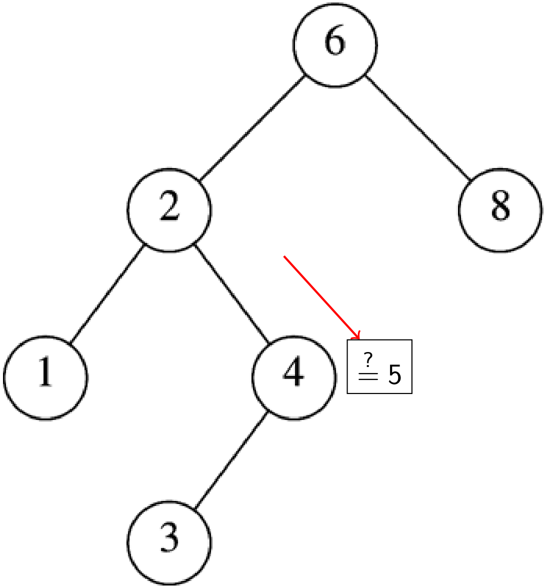
Example Insert



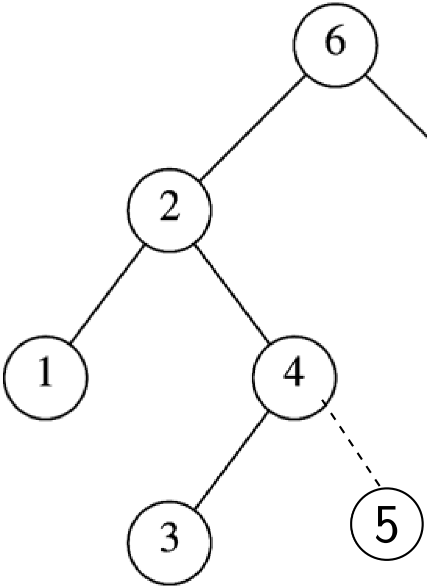
Example Insert



Example Insert



Example Insert



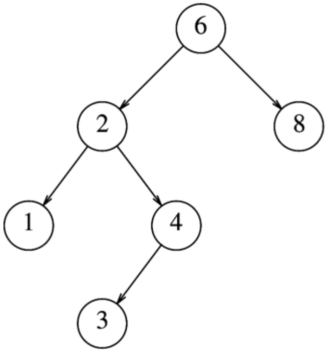
Inserting An Element

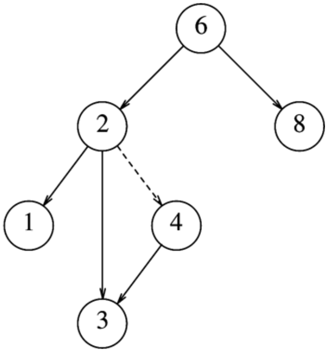
```
template <typename Comparable>
void BST<Comparable>::insert(const Comparable &x) {
    insert(x,root);
}
template <typename Comparable>
void BST<Comparable>::insert(const Comparable &x,Node * &t)
{
    if(t==NULL){
        t=new Node(x,NULL,NULL);
    }
    else if ( x< t->data ){
        insert(x,t->left);
    }
    else{
        insert(x,t->right);
    }
    return ;
}
```

Deleting An Element

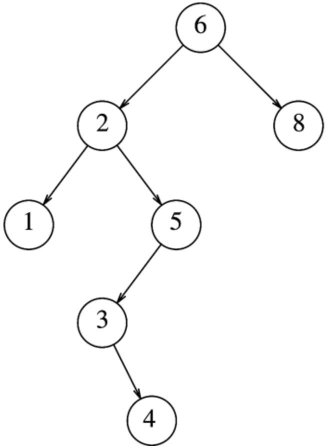
- When deleting a node there are three cases to consider
    1. If the node is a leaf just delete it.
    2. If a node has a single child, that child replaces the node
    3. If a node has two children, the node is replaced by the **smallest** node of the **right subtree** (or the largest node of the left subtree) and that node (which has a maximum of one child) is deleted .

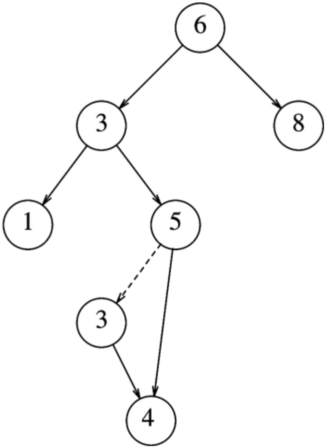
Deleting a node(Value=4) with one child





Deleting a node (Value=2) with two children





Deleting An Element

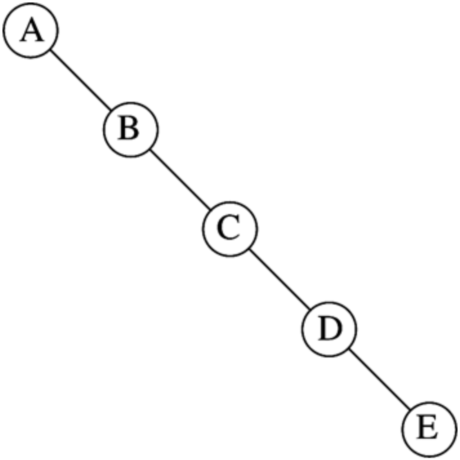
```
template <typename Comparable>
void BST<Comparable>::remove(const Comparable &x,Node * &t){

    if(t==NULL) return;
    if(x<t->data)remove(x,t->left);
    else if (x> t->data) remove(x,t->right);
    /* found the node */
    else if(t->left!=NULL && t->right!=NULL){
        t->data=findMin(t->right)->data;
        remove(t->data,t->right);
    }
    else {
        Node *oldNode=t;
        t=(t->left!=NULL)?t->left:t->right;
        delete oldNode;
    }
}
```

AVL Trees

- An Adelson-Velskii and Landis tree is a binary search tree with a **balance condition**
- We have seen that all operations on a binary search trees of height h are $O(h)$.
- We have also seen that, on **average**, the height of a BST is $\log n$ where n is the number of nodes
- so the **average** complexity of most operations is $O(\log n)$
- For some trees the **worst-case** complexity is $O(n)$

Worst-case Height

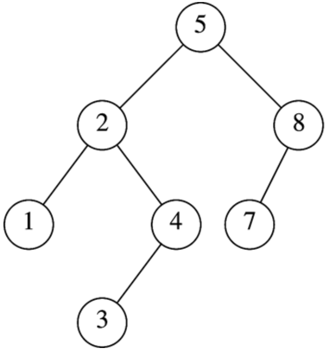


AVL balance condition

- Given a binary search tree and node X.
- Let h_L be the height of the left subtree of X.
- Let h_R be the height of the right subtree of X.
- To make sure that $h = O(\log n)$ we keep the balance condition
- For every node X we always have

$$\mid h_L - h_R \mid \leq 1$$

Example AVL



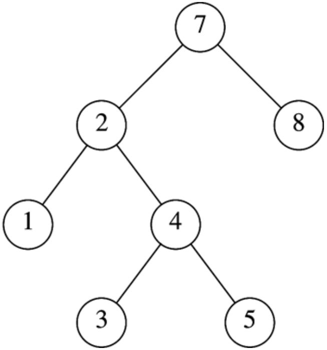


Figure : Two BSTs. Only the left is an AVL tree. In the tree to the right, h(8)=0, h(2)=2

Insertion/Deletion

- Inserting or deleting a node from an AVL tree might destroy the AVL property.
- To restore the AVL property we need to perform **rotations**
- We will study two types of rotations:
  1. Single rotation
  2. Double rotation

Example Insertion

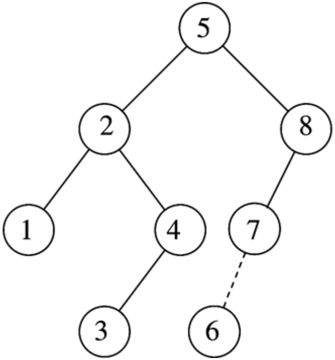


Figure : Inserting Node 6 destroys AVL property

Example Single Rotation

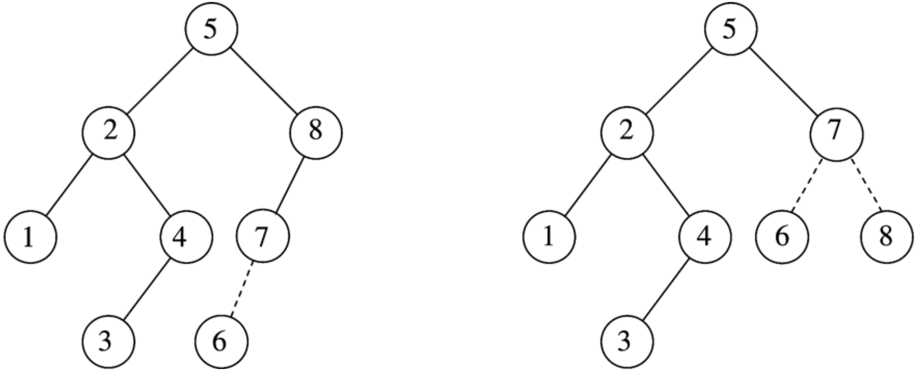
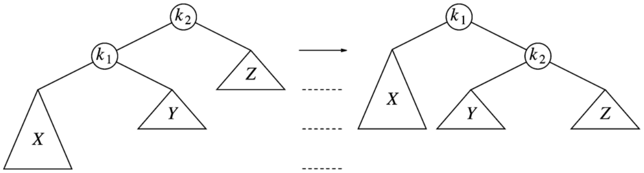
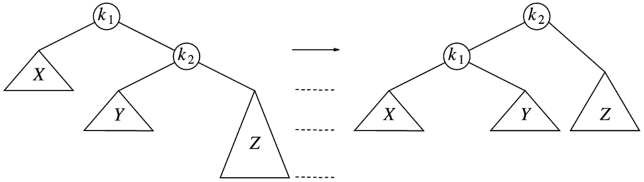


Figure : Example single rotation

Single Rotation



Insertion into the left subtree of the left child



Insertion into the right subtree of the right child

Need for double rotation

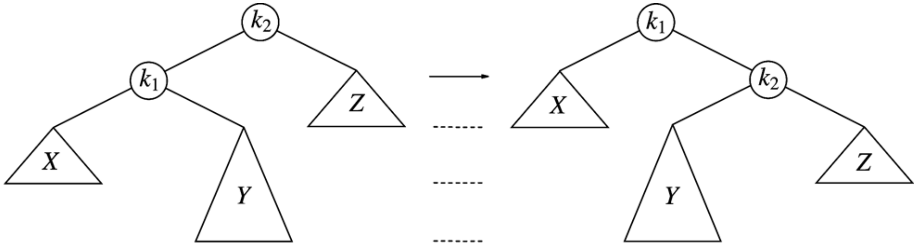
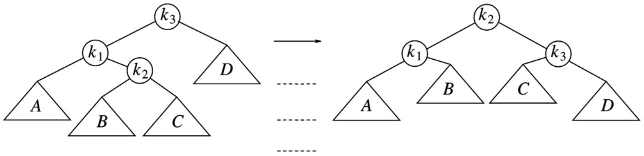
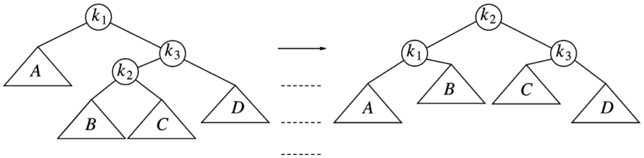


Figure : Single rotation does not work in all cases

Double Rotation



Insertion into the right subtree of the left child



Insertion into the left subtree of the right child

General Rules

- The rules for balancing an unbalanced node n in an AVL trees are as follows
  1. if the unbalance is in the left of the left node L then do a single clock-wise rotation of n.
  2. if the unbalance is in the right of the left node L then do a single counter clock-wise rotation of the left child L followed by a clock-wise rotation of n.
  3. if the unbalance is in the left of the right node R then do a single clock-wise rotation of the right node R followed by a counter clock-wise rotation of n.
  4. if the unbalance is in the right of the right node R then do a single counter clock-wise rotation of n.

Height of tree after rotation

- Consider what happens to a tree after rotation. Suppose that an imbalance was caused by an insertion.
- Let x be the deepest node having imbalance due to the insertion and let y and z be,respectively the roots of the right and left subtrees of x.
- Imbalance at x means the difference in height between y and z is 2.
- Since insertion caused an imbalance then before insertion we had $h_z = h$, $h_y = h + 1$ and $h_x = h + 2$.
- After insertion the height of y increased by 1 but did not cause an imbalance at y then if u and v are the roots of the left and right subtrees of y then before insertion $h_u = h_v = h$.
- The above is illustrated in the figure below

Before Insertion

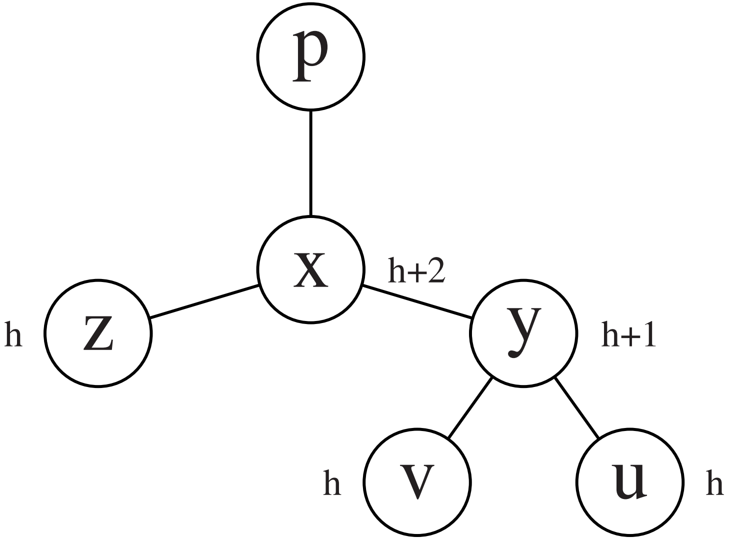


Figure :

- We will show that after rotation (single or double) at node x the tree will have no further imbalance.
- This is done by showing that after rotation (single or double) at node x the height of the child of node p is $h+2$ which is exactly the height of x before insertion.

After Insertion but before rotation,case 1: single rotation

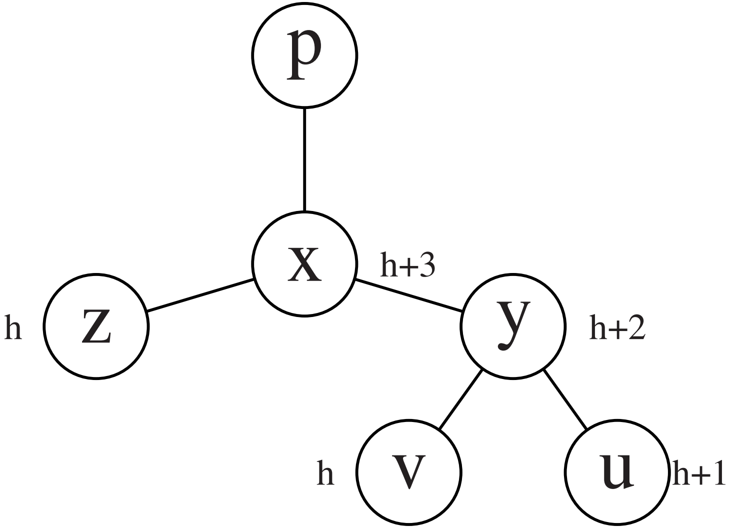


Figure :

After rotation,case 1: single rotation

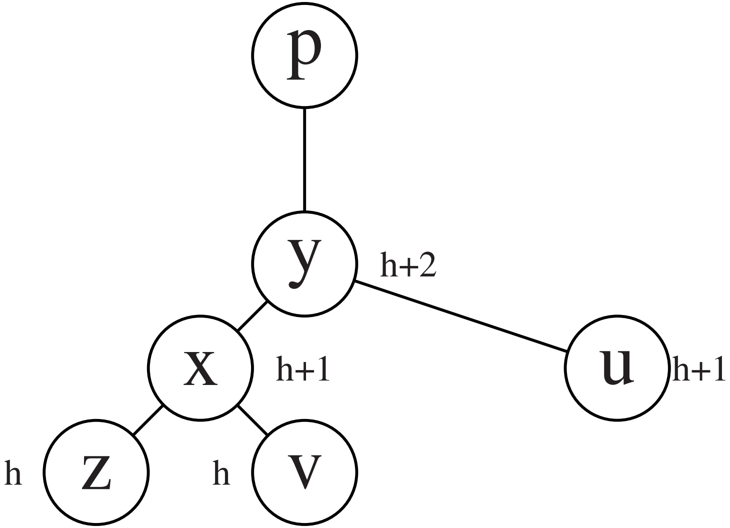
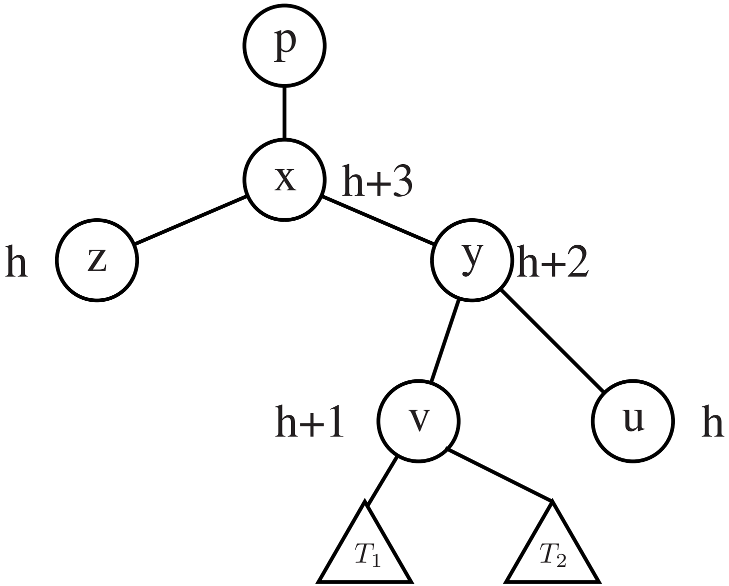


Figure :

After Insertion but before rotation,case 2: double rotation



After Insertion and rotation,case 2: double rotation

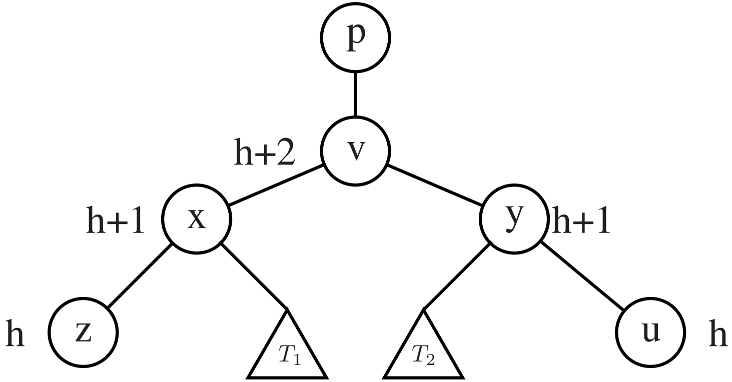


Figure :

Imbalance caused by deletion

- The case of deletion in an AVL tree is different from insertion.
- Whereas for the case of insertion fixing the imbalance at the deepest node will fix all other imbalances as we have shown the case for deletion is different
- Below we show an AVL before deletion, after deletion and after rotation. The example will show that we need to do rotations to the ancestors of the deepest node in addition to the rotation at the node itself.

AVL before deletion

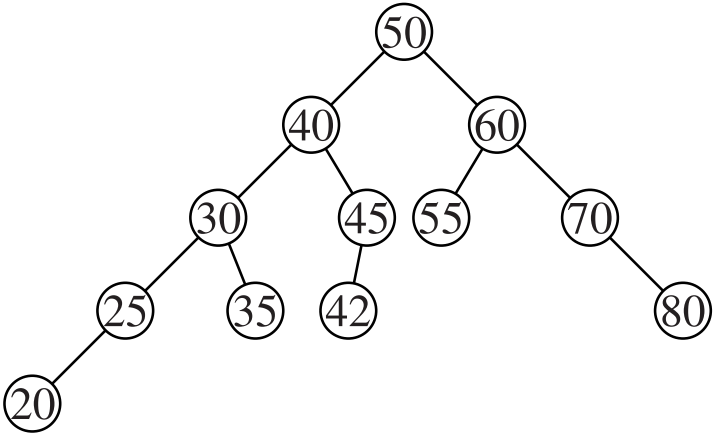


Figure :

- If node 55 is deleted it creates an imbalance at 60 which is the deepest imbalance. A single rotation will fix that imbalance and we obtain the tree below which has an imbalance at 50.

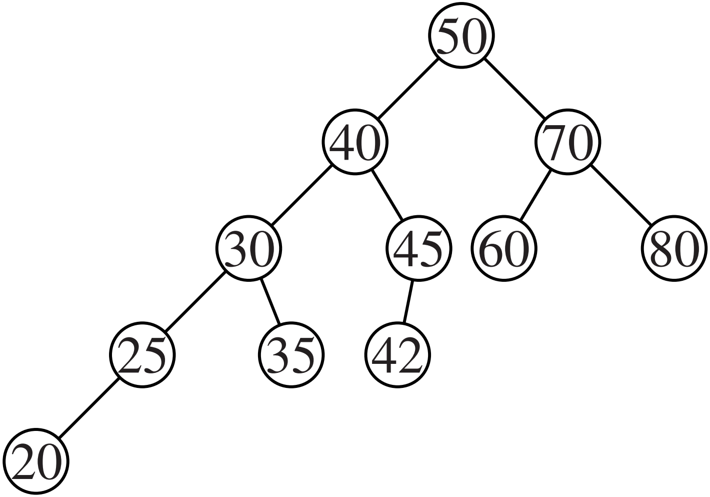


Figure :

Height of AVL trees

- Let T_h be the smallest AVL tree of height h and let N_h be the number of nodes in T_h.
- Let L and R be the left and right subtrees of L_h respectively. Since T_h is the smallest AVL then the height of L and R cannot be the same.
- Then the height of L is $h-2$ and height of R is $h-1$. In addition L is the smallest AVL of height $h-2$ and similarly for R.
- This allow us to obtain the relation

$$N_h = N_{h-2} + N_{n-1} + 1$$

- Adding one to both sides and letting $\mathcal{N}_h = N_h + 1$ we get

$$\mathcal{N}_h = \mathcal{N}_{h-1} + \mathcal{N}_{h-2}$$

- The above is the recurrence relation for the Fibonacci numbers.

- **Note**: while $\mathcal{N}_h$ satisfies the Fibonacci recurrence it does not mean they are the same since the base cases are different.
- For example: $\mathcal{N}_0 = 2, \mathcal{N}_1 = 3$ while $F_0 = 0$ and $F_1 = 1$.
- We can get an upper bound on the height of the minimal AVL using the golden ration $\phi = \frac{1+\sqrt{5}}{2}$

$$\begin{aligned}
\phi^2 &= \left(\frac{1+\sqrt{5}}{2}\right)^2 \\
&= \frac{1+5+2\cdot\sqrt{5}}{4} \\
&= \frac{3+\sqrt{5}}{2} \\
&= 1+\frac{1+\sqrt{5}}{2} \\
&= 1+\phi
\end{aligned}$$

- We prove by induction that $N_h \geq \phi^h - 1$.
- Base case $N_0 = 1 > \phi^0 - 1 = 0$, $N_1 = 2 > \phi^1 - 1 = 1.62 - 1 = 0.62$
- Hypothesis: Assume that $N_h \geq \phi^h - 1$.
- Induction step using the recurrence

$$\begin{aligned} N_{h+1} &= 1 + N_h + N_{h-1} \\ &> \phi^h + \phi^{h-1} - 1 \\ &> \phi^{h-1}(\phi + 1) - 1 \\ &> \phi^{h-1}\phi^2 - 1 \\ &> \phi^{h+1} - 1 \end{aligned}$$

- When N_h is large we drop the -1 term and take the log of both sides

$$\begin{aligned} h\log\phi &< \log N_h \\ h &< \frac{1}{\log\phi}\log N_h \\ h &< 1.44\log N_h \end{aligned}$$

- Since N_h is the smallest number of nodes in AVL of height h then for **any** AVL of height h we have

$$h = O(\log n_h)$$

B-Trees

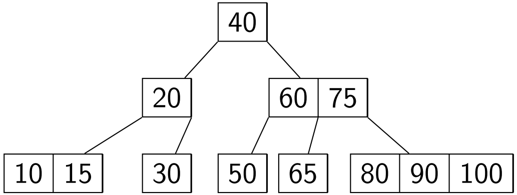
- The need for B-trees arises from the fact that many systems store their data in secondary storage such as disks.
- To see the need for B-trees let us recall how AVL (or BST) search for data while making disk access explicit using a fictitious function read-from-disk

```
bool search(int x,Node *t){
    if(x==t->val)return true;
    if(x<val){
     n=read-from-disk(t->left);
     return search(x,n);
    }
   else {
     n=read-from-disk(t->right);
     return search(x,n);
   }
}
```

B-Trees

- Since CPU operations (like comparison) are order of magnitudes faster than disk access we would like to minimize disk access even at the expense of more CPU operations
- The above strategy is what makes B-Trees attractive, minimize the height of the tree (which minimizes disk access) at the expense of more comparisons (faster operations).
- A B-Tree stores more data per node at the expense of extra CPU operations.

Example B-Tree



B-Trees

- Every node in a B-tree has a variable number of keys (see below) stored in a non-decreasing order.
- If an internal node has n keys it would have $n+1$ pointers to children.
- All leaves have the same depth which is equal to the height of the tree.
- The range of the number of keys is determined by the *minimum degree* of the tree t.
  1. Every node other than the root has at least $t-1$ keys and at most $2t-1$ keys.
  2. The above means it has at least t children and at most $2t$ children.

Searching in B-tree

- Since the number of keys is variable it is stored in the node. Given node a then $a.n$ is the number of keys.
- The search for value x in a B-tree is done recursively.
- We start at the root and compare x with the keys (left to right) stored in the root node
- if x matches any key then it is found.
- If $k_i < x < k_{i+1}$ then the next node to consider is the one pointed to by c_{i+1}.

B-tree Search

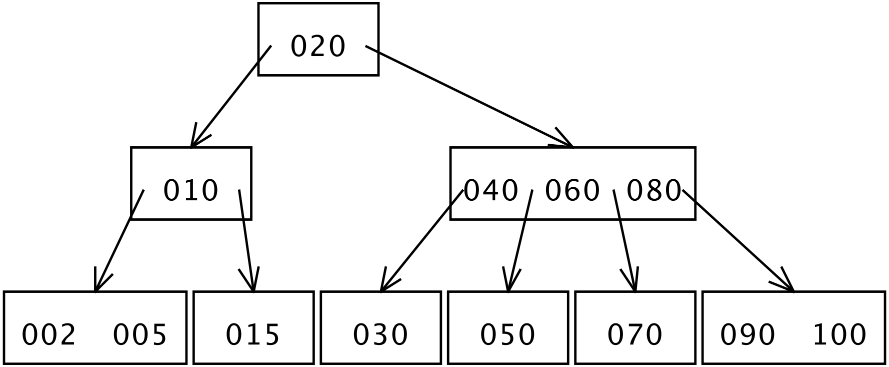
```
B-T-Search (a,x)
$i \leftarrow 1$
while $i \leq a.n$ and $x > a.k_i$ do
    $i \leftarrow i + 1$
if $i \leq a.n$ and $x = a.k_i$ then
    return True
else if $a.leaf$ then
    return False
else
    Read-disk($a.c_i$)
    return B-T-Search ($a.c_i$,$x$)
```

B-Tree Insertion

- Insertion into a B-tree is much more complicated than a BST
- Insertion of values in a B-Tree is done at the leaves.
- To find the proper leaf a recursive descent, starting from the root, is performed as in the case of search.
- While descending the tree if a node is full it is split in two.

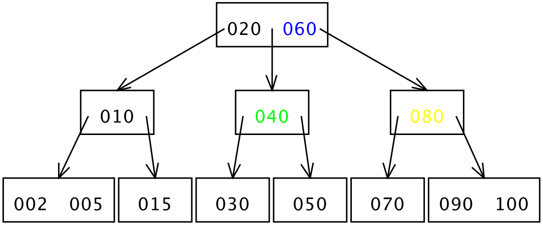
B-Tree Insertion

- Consider the insertion of the value 55 in the tree below

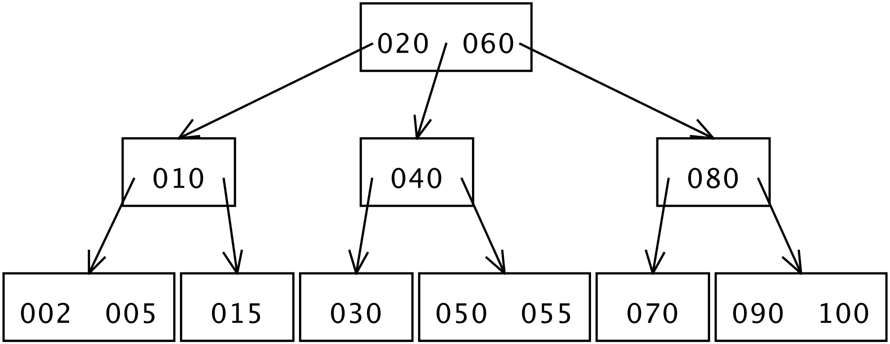


- Starting from the root we compare $55 > 20$ so the next node to consider is [40,60,80]
- Since $40 < 55 < 60$ then the value 50 will be inserted in the leaf [50].
- But before descending to node $[40, 60, 80]$, which is full, it is split into two and then the descent continues

Inserting: 055



- After splitting the node $[40, 60, 80]$ in two the value 55 is inserted in the appropriate leaf as shown below



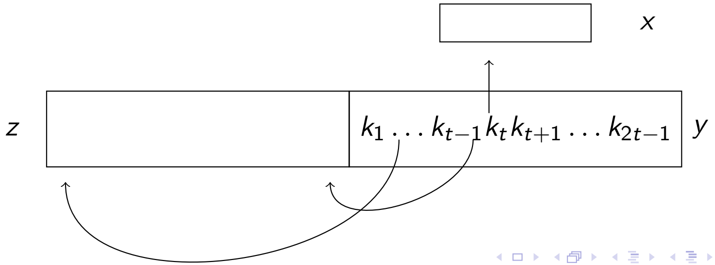
Node Splitting

- A node is split when it is full, i.e. number of keys$= 2t - 1$.
- The median key at index t will be "promoted" to the parent node
- The other keys: from index 1 to $t-1$ are moved to a newly created node and the remaining keys from $t+1$ to $2t-1$ remain in the original node

y is a full child of x with $2t-1$ keys.

$t-1$ keys are moved to a newly created node z.

and key k_t is moved to x.



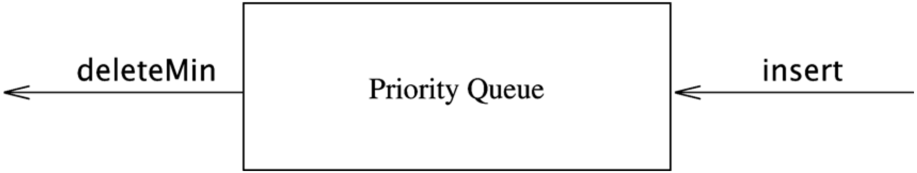
Node splitting

```
B-T-Split-Child (x,i)/* split child i of node x                    */

$i \leftarrow 1$
while $i \leq a.n$ and $x > a.k_i$ do
    $i \leftarrow i + 1$
if $i \leq a.n$ and $x = a.k_i$ then
    return True
else if $a.leaf$ then
    return False
else
    Read-disk($a.c_i$)
    return B-T-Search ($a.c_i$,$x$)
```

Priority Queue

- A priority queue is an ADT that allows for at least two operations
  1. **insert** to add an element to the queue which is equivalent to enqueue
  2. deleteMin (or deleteMax) which finds and returns and deletes the minimum (or maximum) value. This is equivalent to the dequeue operation.



Simple Implementation

- A simple implementation of a priority queue can be done using a linked list
  1. insertion is done at the head (or tail if we keep track of it) of the list in $O(1)$
  2. deleteMin can be done in $O(n)$.
- A better way would be to use a binary Heap.

Binary Heap

- A binary heap is a binary tree of height h where the subtree of height $h-1$ is perfect.
- This means that all levels, with the possible exception of the bottom one, are completely filled.

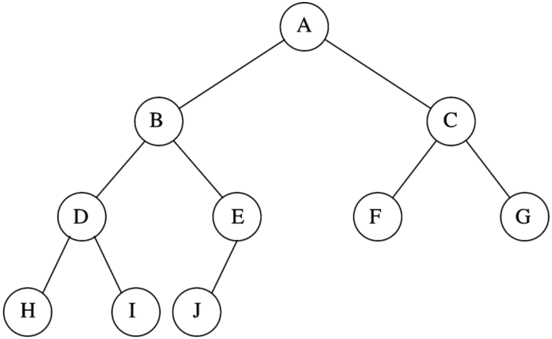
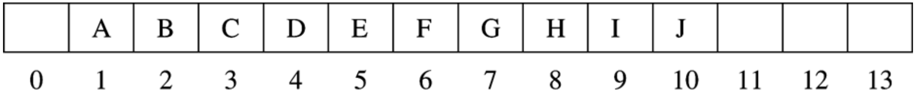


Figure : Binary Heap Example

- It is easily shown that the number of nodes n of a binary heap of height h is $2^h \leq n \leq 2^{h+1} - 1$ nodes
- This means that all operations are $O(\log n)$.
- Because of this regularity of a binary heap it can be represented as an array
- For node at element i its left child is at $2i$ and right child is at $2i+1$ and its parent is at $\lfloor i/2 \rfloor$



	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Figure : Array representation of binary heap

Heap Order Property

- To implement a priority queue efficiently we require that our structure obeys the heap order property

In a heap, for every node X the value of X is smaller than the value of its children

- If this property is true for all nodes than the minimum is the root

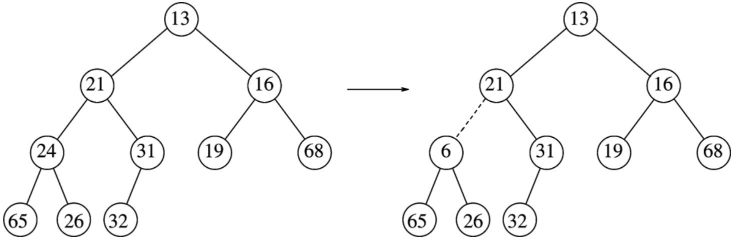
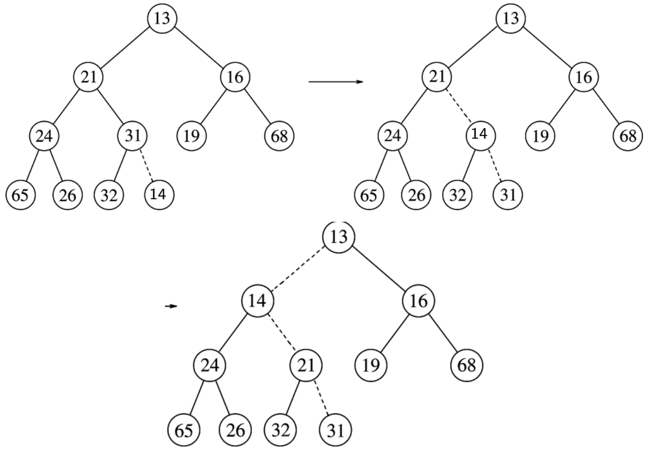


Figure : Left is a heap. Right is not

Inserting an Element

- Obviously inserting an element can destroy the heap property.
- To keep the heap property after insertion we add an element in two steps:
  1. The element is inserted at the first empty position
  2. We recursively compare the inserted element with its parents:
     - ★ If it is smaller it is swapped with its parent.
     - ★ otherwise we stop.
  3. The above operation, which is called percolated up, has a worst-case complexity of $O(h)$.
  4. Since in a binary heap $h = \Theta(\log n)$ then the complexity is $O(\log n)$.
  5. As an example in the next slides we insert element with value 14.

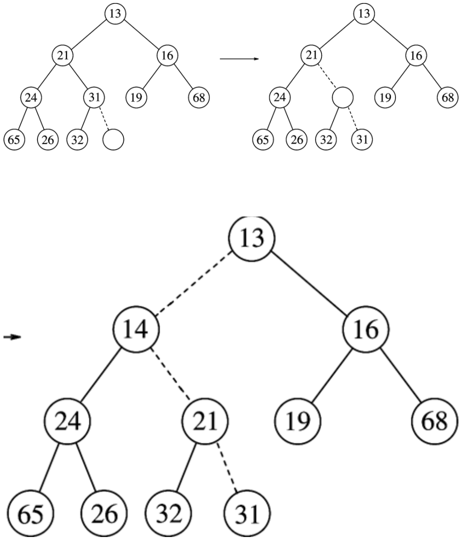
Example


13
21
16
24
31
19
68
65
26
32
14
13
21
16
24
14
19
68
65
26
32
31
13
14
16
24
21
19
68
65
26
32
31

Implementation

```
void insert(int val){
   A[++size]=val;
   int hole=size;
   while(hole/2>=1){
      if(val<A[hole/2]){
         A[hole]=A[hole/2];
         hole=hole/2;
      }
      else
         break;
   }
   A[hole]=val;
}
```

Heap Insert Example



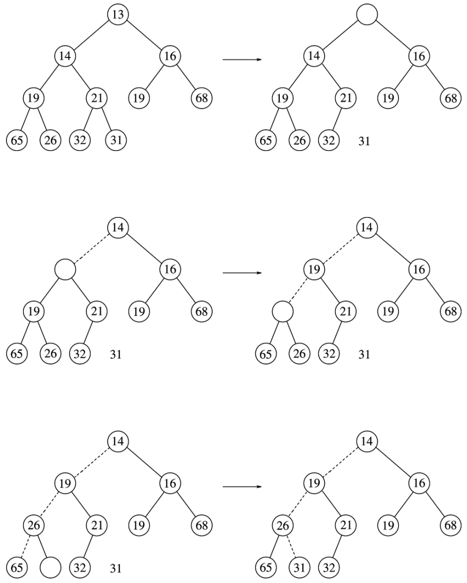
Deleting Minimum

- As mentioned before one of the min-heap operations is deleteMin() which returns, and deletes, the minimum of the heap.
- This will leave a hole in the root of the tree.
- By the min-heap property the root of **any** subtree contains the minimum of that subtree.
- Therefore when the root is removed the minimum of its two children will replace it.
- This will leave a hole in the place of the chosen child.
- Doing the above recursively will make the hole trickle, or percolate, down.

DeleteMin: Take 1

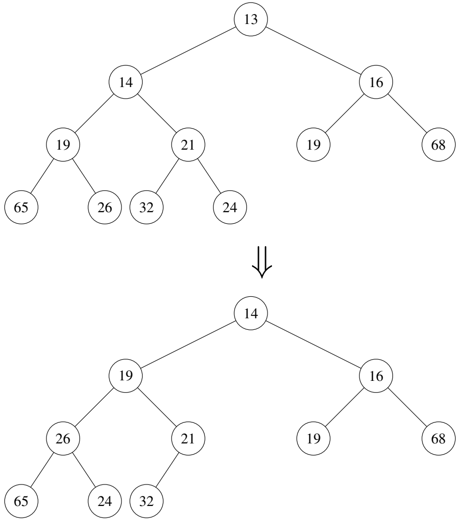
```
hole=1;
while(2*hole<=size){
 child=2*hole;
 if(A[child+1]<A[child]) child++;
A[hole]=A[child];
 hole=child;
}
```

- The above code will leave a hole in one of the leaves which breaks the heap property.
- We can fill it with the last element of the heap and reduce the size by one. Does it work?


13 14 16 19 21 19 68 65 26 32 31
14 16 19 21 19 68 65 26 32 31
14 16 19 21 19 68 65 26 32 31
14 19 16 21 19 68 65 26 32 31
14 19 16 26 21 19 68 65 32 31
14 19 16 26 21 19 68 65 31 32

Does it always work?

- No! Consider the case below and the result when 13 is deleted



DeleteMin: Take 2, percolateDown

- To fix the problem we need to check if the "promoted" value is larger than the last value.

```
hole=1;
last=A[size];

while(2*hole<=size){
 child=2*hole;
 if(A[child+1]<A[child]) child++;
 if(A[child]<last){
  A[hole]=A[child];
  hole=child;
 }
 else{
   A[hole]=last;
   break;
 }
}
```

Building a Heap

- One way of building a heap is by using repeatedly inserting elements.
- Since insertion is $O(\log n)$ then to insert n elements it takes $O(n \log n)$.
- We can do better if we need to build a heap with n elements without any operations in between.
- The idea is first to add the elements in an unordered way.
- Then use the percolateDown() method defined earlier to reorder the heap.

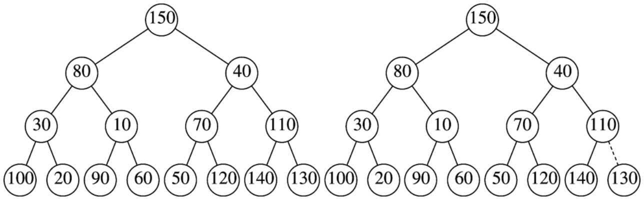


Figure : Left: initial heap. Right:after percolate(7)

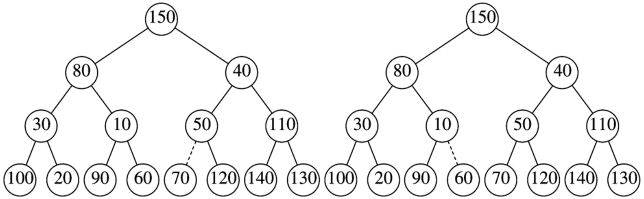


Figure : Left: after percolate(6). Right:after percolate(5)

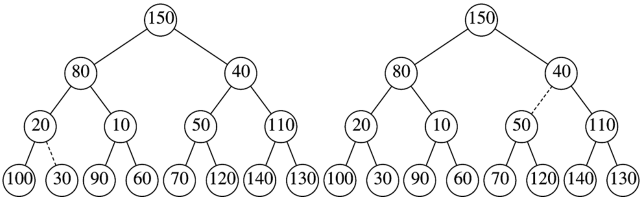


Figure : Left: after percolate(4). Right:after percolate(3)

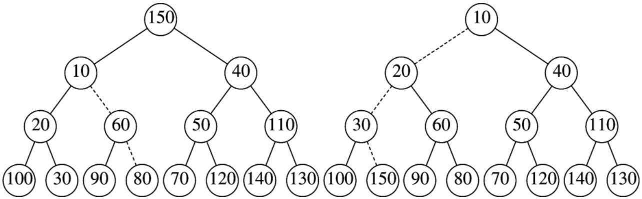


Figure : Left: after percolate(2). Right:after percolate(1)

Implementation

```
BinaryHeap(vector<int> & items)
:A(items.size()+1),size(items.size()){
  for(int i=0;i<items.size();i++)
     A[i+1]=items[i];

 buildHeap();
}

void buildHeap(){
  for(int i=size/2;i>0;i--)
    percolateDown(i);
}
```

Complexity of BuildHeap

- Is build heap more efficient than using insert?
- We will show that building the heap this way is $O(n)$ whereas using insert as we have seen is $O(n \log n)$.
- First observe that the complexity of each percolation operation is bounded by the height of the starting node.
- Therefore the total complexity of building a heap of height h is the sum of the heights of all nodes.

- In a full binary tree the number of nodes at depth d is 2^d.
- on the other hand, a node with depth d has height $h-d$
- therefore the sum of the height of all nodes is

$$\begin{aligned} H &= \sum_{d=0}^{h} 2^d (h-d) \\ &= h + 2(h-1) + 4(h-2) + 8(h-3) + \ldots + 2^{h-1}\cdot 1 \end{aligned}$$

- Multiplying the above by 2 and compare H and $2H$

$$\begin{array}{l} H = h + 2(h-1) + 4(h-2) + 8(h-3) + \ldots + 2^{h-1}\cdot 1 \\ \qquad\qquad \nearrow \qquad\qquad \nearrow \qquad\qquad \nearrow \\ 2H = 2h + 4(h-1) + 8(h-2) + 16(h-3) + \ldots + 2^{h-1}\cdot 2 + 2^h\cdot 1 \end{array}$$

- By subtracting the expression of H from $2H$ we get

$$\begin{aligned} H &= -h + 2 + 4 + 8 + \ldots + 2^{h-1} + 2^h \\ &= -(h+1) + (1 + 2 + 4 + \ldots + 2^h) \\ &= -(h+1) + (2^{h+1} - 1) \end{aligned}$$

- Therefore the complexity of build heap is $O(2^h) = O(n)$

Application: k^{th} smallest element

- Suppose that we have n numbers and we would like to find the k^{th} smallest.
- One way is to build a heap with n numbers with complexity $O(n)$ and call deleteMin k times with a complexity of $k \log n$.
- The total complexity is $O(n + k \log n)$ which depends on k.
- If k is small compared to n, i.e. $k = O(n/\log n)$ then the complexity is $O(n)$
- other wise the complexity is $O(n \log n)$.
- Finally if $k = n$ and we store the results of deleteMin in, say, an array, we would have sorted n elements in $O(n \log n)$.