Heaps with STL

Complexity: O(log n)

```
vector<int> v;
v.push_back(randomNum); cout << v[i]
  << " ";
make_heap(v.begin(), v.end());
v.push_back(avgTotal); push_heap(v.
  begin(), v.end()); //add vector
for (size_t i=0; i<v.size(); i++)
cout << "Max" << v.front();
pop_heap(v.begin(), v.end());
v.pop_back();
sort_heap(v.begin(), v.end());
find-max/min, delete/extract-max/min,
  replace, insert, heapify, merge(k)/
  meld(r), isHeap, reverse
```

Hashing

```
bool search(int X) {
- if (X >= 0)
-  if (has[X][0] == 1)
-   return true;
- X = abs(X);
-  if (has[X][1] == 1)
-   return true; }
void insert(int a[], int n)
- for (int i = 0; i < n; i++)
-  if (a[i] >= 0)
-   has[a[i]][0] = 1;
-  else
-   has[abs(a[i])][1] = 1; }
int main()
- int a[] = { -1, 9, -5, -8 }
- int n=sizeof(a)/sizeof(a[0]);
- insert(a, n);
- int X = -5;
- if (search(X) == true)
-  cout << "Present";
```

TSP

```
#include <bits/stdc++.h>
int TSP(int graph[][V], int s)
- vector<int> vertex;
- for (int i = 0; i < V; i++)
-  if (i != s)
-   vertex.push_back(i);
- int min_path = INT_MAX;
- do {
-  int current_pathweight = 0;
-  int k = s;
-  for (int i = 0; i < vertex.size();
  i++)
-   current_pathweight += graph[k]
  [vertex[i]];
-   k = vertex[i];
-  current_pathweight += graph[k][s];
-  min_path = min(min_path, current_
  pathweight); }
- while (next_permutation(vertex.
  begin(), vertex.end()));
```

Dijkstra

Complexity: O(n^2)

```
int minDistance(int dist[], bool
  sptSet[])
- for (int v = 0; v < V; v++)
-  if (sptSet[v] == false && dist[v]
  <= min)
-   min = dist[v], min_index=v
- return min_index;
void dijkstra(int graph[V][V], int
  src)
- int dist[V];
- bool sptSet[V];
- for (int i = 0; i < V; i++)
-  dist[i] = INT_MAX, sptSet[i] =
  false;
- dist[src] = 0;
- for (int count = 0; count < V - 1;
  count++)
-  int u = minDistance(dist, sptSet);
-  sptSet[u] = true;
-  for (int v = 0; v < V; v++)
-   if (!sptSet[v] && graph[u][v] &&
  dist[u] != INT_MAX && dist[u] +
  graph[u][v] < dist[v])
-   dist[v] = dist[u] + graph[u][v];
```

Priority Queue

```
q.top(), q.pop(), q.push(n), q.size()
while(!q.empty())
  print g.top, g.pop
prime for loop (i=2; n/2; i++)
```

Merge Sort

```
void merge(int arr[], int l, int m,
  int h)
- int i, j, k;
- int n1 = m - l + 1;
- int n2 =  h - m;
- int L[n1], R[n2];
- for (i = 0; i < n1; i++)
-  L[i] = arr[l + i];
- for (j = 0; j < n2; j++)
-  R[j] = arr[m + 1+ j];
- i = 0; j = 0; k = l;
- while (i < n1 && j < n2)
-  if (L[i] <= R[j])
-   arr[k] = L[i]; i++;
-  else
-   arr[k] = R[j]; j++;
-  k++;
- while (i < n1)
-  arr[k] = L[i]; i++; k++;
- while (j < n2)
-  arr[k] = R[j]; j++; k++;
void mergeSort(int arr[], int l, int
  h)
- if (l < h)
-  int m = l+(h-l)/2;
-  mergeSort(arr, l, m);
-  mergeSort(arr, m+1, h);
-  merge(arr, l, m, h);
```

Heap Sort

```
void heapify(int arr[], int n, int i)
- int largest = i;
- int l = 2*i + 1;
- int r = 2*i + 2;
- if (l < n && arr[l] > arr[largest])
-  largest = l;
- if (r < n && arr[r] > arr[largest])
-  largest = r;
- if (largest != i)
-  swap(arr[i], arr[largest]);
-  heapify(arr, n, largest);
void heapSort(int arr[],int n)
- for (int i=n/2-1; i>=0; i--)
-  heapify(arr, n, i);
- for (int i=n-1; i>=0; i--)
-  swap(arr[0], arr[i]);
-  heapify(arr, i, 0);
```

RBT Fix-Up

```
RedBlackNode<ItemType>* LeftLeanin-
  gRedBlackTree<ItemType>::fixUp(Red-
  BlackNode<ItemType> *subTreePtr)
- if (isRed(subTreePtr-
  >getRightChildPtr()))
-  subTreePtr =
  rotateLeft(subTreePtr);
- if (isRed(subTreePtr-
  >getLeftChildPtr()) &&
  isRed(subTreePtr->getLeftChildPtr()-
  >getLeftChildPtr()))
-  subTreePtr =
  rotateRight(subTreePtr);
- if (isRed(subTreePtr-
  >getLeftChildPtr())
  && isRed(subTreePtr-
  >getRightChildPtr()))
-  colorFlip(subTreePtr);
- return subTreePtr;
```

MST/K-th Smallest Element

```
int ksmallestElementSumRec(Node *root,
  int k, int &count)
- if (root == NULL)
-  return 0;
- if (count > k)
-  return 0;
- int res = ksmallestElementSumRec(-
  root->left, k, count);
- if (count >= k)
-  return res;
- res += root->data;
- count++;
- if (count >= k)
-  return res;
- return res + ksmallestElementSum-
  Rec(root->right, k, count);
```

RBT Insert

```
RedBlackNode<ItemType>* LeftLean-
  ingRedBlackTree<ItemType>::inser-
  tRec(RedBlackNode<ItemType> *sub-
  TreePtr, RedBlackNode<ItemType>
  *newNodePtr)
- RedBlackNode<ItemType> *tempPtr =
  nullptr;
- if (subTreePtr == nullptr)
-  return newNodePtr;
- if (subTreePtr->getItem() > newN-
  odePtr->getItem())
-  subTreePtr->setLeftChildPtr(inser-
  tRec(subTreePtr->getLeftChildPtr(),
  newNodePtr));
- else
-  subTreePtr->setRightChildPtr(inser-
  tRec(subTreePtr->getRightChildPtr(),
  newNodePtr));
- if (isRed(subTreePtr->get-
  RightChildPtr()) && !isRed(sub-
  TreePtr->getLeftChildPtr()))
-  subTreePtr = rotateLeft(sub-
  TreePtr);
- if (isRed(subTreePtr->ge-
  tLeftChildPtr()) && isRed(sub-
  TreePtr->getLeftChildPtr()->ge-
  tLeftChildPtr()))
-  subTreePtr = rotateRight(sub-
  TreePtr);
- if (isRed(subTreePtr->ge-
  tLeftChildPtr()) && isRed(sub-
  TreePtr->getRightChildPtr()))
-  colorFlip(subTreePtr);
- return subTreePtr;
```

RBT Rotate Left

```
RedBlackNode<ItemType>* LeftLean-
  ingRedBlackTree<ItemType>::ro-
  tateLeft(RedBlackNode<ItemType>
  *subTreePtr)
- RedBlackNode<ItemType> *tempPtr =
  subTreePtr->getRightChildPtr();
- subTreePtr->setRightChildPtr(-
  tempPtr->getLeftChildPtr());
- tempPtr->setLeftChildPtr(sub-
  TreePtr);
- tempPtr->setRed(sub-
  TreePtr->getRed());
- subTreePtr->setRed(1);
- return tempPtr;
```

Depth-First Search

The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process.

```
void Graph::addEdge(int v, int w)
- adj[v].push_back(w);
void Graph::DFSUtil(int v, bool vis-
  ited[])
- visited[v] = true;
- cout << v << " ";
- list<int>::iterator i;
- for (i = adj[v].begin(); i != ad-
  j[v].end(); ++i)
-  if (!visited[*i])
-   DFSUtil(*i, visited);
void Graph::DFS(int v)
- bool *visited = new bool[V];
- for (int i = 0; i < V; i++)
-  visited[i] = false;
- DFSUtil(v, visited);
```

Breadth-First Search

```
void Graph::addEdge(int v, int w)
- adj[v].push_back(w);
void Graph::BFS(int s)
- bool *visited = new bool[V];
- for(int i = 0; i < V; i++)
-  visited[i] = false;
- list<int> queue;
- visited[s] = true;
- queue.push_back(s);
- list<int>::iterator i;
- while(!queue.empty())
-  s = queue.front();
-  cout << s << " ";
-  queue.pop_front();
- for (i = adj[s].begin(); i != ad-
  j[s].end(); ++i)
-  if (!visited[*i])
-   visited[*i] = true;
-   queue.push_back(*i);
```

Prim's Algorithim

Start at any vertex, look for shortest on any visited vertex, without cycles.

Kruskal's MST

Start at the smallest cost, keep ascending up without cycle.

```
void KruskalMST(Graph* graph)
- int V = graph->V;
- Edge result[V];
- int e = 0;
- int i = 0;
- qsort(graph->edge, graph->E, sizeo-
  f(graph->edge[0]), myComp);
- subset *subsets = new subset[( V *
  sizeof(subset) )];
- for (int v = 0; v < V; ++v)
-  subsets[v].parent = v;
-  subsets[v].rank = 0;
- while (e < V - 1 && i < graph->E)
-  Edge next_edge = graph->edge[i++];
-  int x = find(subsets, next_edge.
  src);
-  int y = find(subsets, next_edge.
  dest);
-  if (x != y)
-   result[e++] = next_edge;
-   Union(subsets, x, y);
- cout<<"Following are the edges in
  the constructed MST\n";
- for (i = 0; i < e; ++i)
-  cout<<result[i].src<<" -- "<<re-
  sult[i].dest<<" == "<<result[i].
```

```
  weight<<endl;
- return;
```

BST Destroy Tree / Traversals

```
Remember: if (rootPtr != nullptr) { }
void BinarySearchTree<ItemType>::de-
  stroyTree(BinaryNode<ItemType>
  *subTreePtr)
- if (subTreePtr != nullptr)
-  destroyTree(subTreePtr->ge-
  tLeftChildPtr());
-  destroyTree(subTreePtr->get-
  RightChildPtr());
-  subTreePtr.reset();
Preorder: C;P;P;
Inorder: I;C;I;
Postorder: P;P;C;
```

BST Remove Value

```
auto BinarySearchTree<ItemType>::re-
  moveValue(BinaryNode<ItemType> *sub-
  TreePtr, const ItemType target, bool
  &isSuccessful)
- BinaryNode<ItemType> *tempPtr =
  nullptr;
- if (subTreePtr == nullptr)
-  isSuccessful = 0;
- else if (subTreePtr->getItem() ==
  target)
-  subTreePtr = removeNode(sub-
  TreePtr);
-  isSuccessful = 1;
- else if (subTreePtr->getItem() >
  target)
-  tempPtr = removeValue(sub-
  TreePtr->getLeftChildPtr(), target,
  isSuccessful);
-  subTreePtr->setLeftChildPtr(-
  tempPtr);
- else
-  tempPtr = removeValue(sub-
  TreePtr->getRightChildPtr(), target,
  isSuccessful);
-  subTreePtr->setRightChildPtr(-
  tempPtr);
- return subTreePtr;
```

BST Remove

```
bool BinarySearchTree<ItemType>::re-
  move(const ItemType &target)
- bool isSuccessful = 0;
- this->rootPtr = removeVal-
  ue(this->rootPtr, target, isSuccess-
  ful);
- return isSuccessful;
```

BST Copy Tree

```
auto BinarySearchTree<ItemType>::-
  copyTree(const BinaryNode<ItemType>
  *oldTreeRootPtr) const
- BinaryNode<ItemType> *newTreePtr =
  nullptr;
- if (oldTreeRootPtr != nullptr)
-  newTreePtr = new BinaryNode<Item-
  Type>(oldTreeRootPtr->getItem(),
  nullptr, nullptr);
-  newTreePtr->setLeftChildPtr(-
  copyTree(oldTreeRootPtr->ge-
  tLeftChildPtr()));
-  newTreePtr->setRightChildPtr(-
  copyTree(oldTreeRootPtr->get-
  RightChildPtr()));
- return newTreePtr;
```

BST Add

```
bool BinarySearchTree<ItemType>::add(-
  const ItemType &newData)
- BinaryNode<ItemType> *newNodePtr =
  new BinaryNode<ItemType>(newData);
- this->rootPtr = placeNode(rootPtr,
  newNodePtr);
- return 1;
```

BST Place Node

```
auto BinarySearchTree<ItemType>::-
  placeNode(BinaryNode<ItemType>
  *subTreePtr, BinaryNode<ItemType>
  *newNodePtr)
- BinaryNode<ItemType> *tempPtr =
  nullptr;
- if (subTreePtr == nullptr)
-  return newNodePtr;
- else if (subTreePtr->getItem() >
  newNodePtr->getItem())
-  tempPtr = placeNode(subTreePtr->ge-
  tLeftChildPtr(), newNodePtr);
-  subTreePtr->setLeftChildPtr(-
  tempPtr);
- else
-  tempPtr = placeNode(sub-
  TreePtr->getRightChildPtr(), newN-
  odePtr);
-  subTreePtr->setRightChildPtr(-
  tempPtr);
- return subTreePtr;
```

Binary Search Tree

The depth of a node is the number of edges from the root to the node.
The height of a node is the number of edges from the node to the deepest leaf.
The height of a tree is a height of the root.
A full binary tree is a binary tree in which each node has exactly zero or two children.
A complete binary tree is a binary tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right.
h = O(log n)
Search/Insert/Delete = avg: O(log n)
worst: O(n)

AVL/Balanced Search Tree

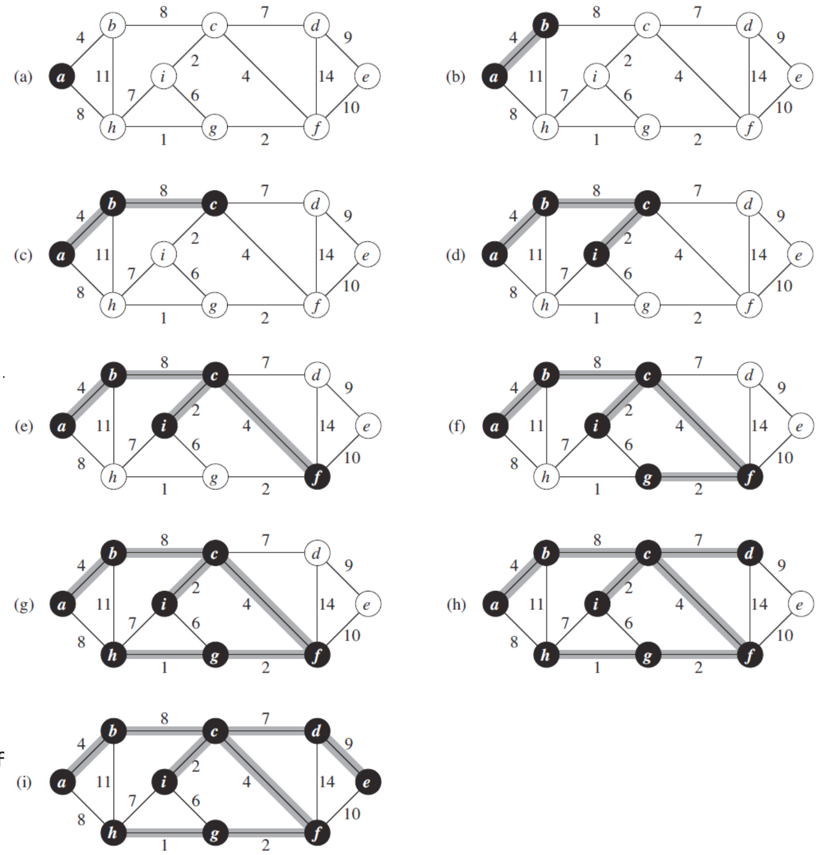
Height difference between left and right node cannot exceed 1
Search/Insert/Delete = O(logn)

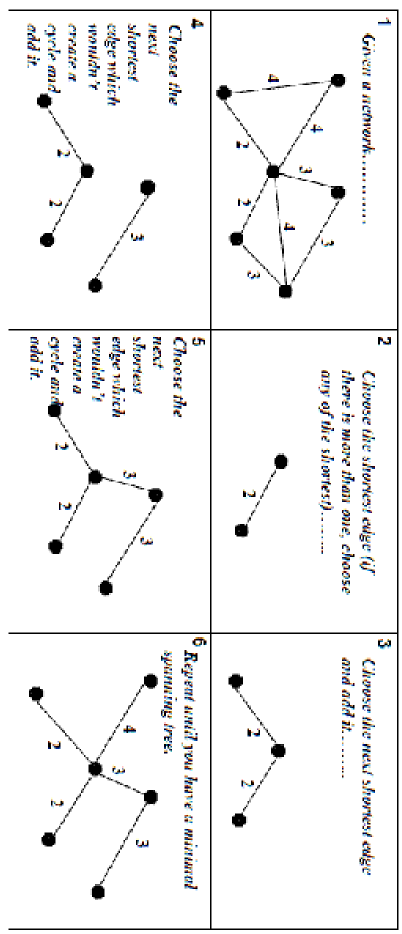
Red Black Tree

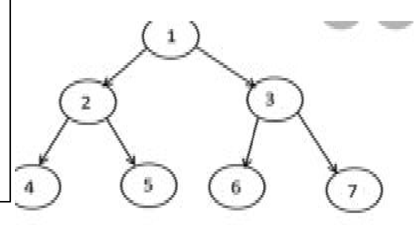
Every node has a color either red or black.
Root of tree is always black.
There are no two adjacent red nodes (A red node cannot have a red parent or red child).
h = O(log n)
Search/Insert/Delete = O(logn)
Color Change = O(1) * height

2-3-4 Trees

Insertion into a 2-3-4 Tree begins with a single node where values are inserted until it becomes full (ie. until it becomes a 4-node). The next value that is inserted will cause a split into two nodes: one containing values less than the median value and the other containing values greater than the median value. The median value is then stored in the parent node. It's possible that insertion could cause splitting up to the root node.







Inorder Traversal: 4 2 5 1 6 3 7
Preorder Traversal: 1 2 4 5 3 6 7
Postorder Traversal: 7 6 3 5 4 2 1
Breadth-First Search: 1 2 3 4 5 6 7
Depth-First Search: 1 2 4 5 3 6 7