Topics List for Exam 2

- **Queues** (FIFO) #include <queue> **can't use vectors or arrays with queues!!!**
  - *Know how to create an empty queue*

```
deque<int> mydeck (3,100);  // deque with 3 elements
list<int> mylist (2,200);     // list with 2 elements

queue<int> first;     // empty queue
queue<int> second (mydeck);   // queue initialized to copy of deque

queue<int,list<int> > third; // empty queue with list as underlying container
queue<int,list<int> > fourth (mylist);

cout << "size of first: " << first.size() << '\n';
cout << "size of second: " << second.size() << '\n';
cout << "size of third: " << third.size() << '\n';
cout << "size of fourth: " << fourth.size() << '\n';
```

  - *know what push(), pop(), empty(), and size() do.* (front(), end())

Void **push(**T value) adds to the back of the queue

```
myqueue.push (myint);
```

Void **pop()** removes from the front of the queue

```
myqueue.pop();
```

Bool **empty()** returns true or false

```
while (!myqueue.empty())
 {
        sum += myqueue.front();
        myqueue.pop();
 }
```

Int **size()** returns the size of the queue

```
for (int i=0; i<5; i++) myints.push(i);
cout << "1. size: " << myints.size();
```

- **Priority Queues (**ts first element is always the greatest of the elements it contains)
  - *Know how to create an empty priority_queue.*

```
priority_queue<int> first; //empty
priority_queue<int> second (myints,myints+4);
priority_queue<int, std::vector<int>, std::greater<int> >third (myints,myints+4);
```

  - *Know what push(), pop(), top(), size() and empty() do, and be able to use them to write a function*

Void **push(**const int value & val) This member function effectively calls the member function push_back of the *underlying container* object, and then reorders it to its location in the *heap* by calling the push_heap algorithm on the range that includes all the elements of the container.

Void **pop()** Removes the element on top of the priority_queue, effectively reducing its size by one. The element removed is the one with the highest value.

T **size()**; Returns the number of elements in the priority_queue.

T **top()**; Returns a constant reference to the *top element* in the priority_queue.

Example:

```
priority_queue<int> mypq;
  mypq.push(30);
  mypq.push(100);
  mypq.push(25);
  mypq.push(40);

  cout<<mypq.size();

  while (!mypq.empty()) {
        cout << ' ' << mypq.top();//prints top element
        mypq.pop(); //removes an element
  }
```

- Be able to solve a problem using a priority queue.
- Our own implementation:
  - *Understand the implementation we used (binary max heap.)* (the max value is the root of the tree) (http://www.sci.brooklyn.cuny.edu/~mermelstein/Courses/CISC3130/PriorityQueue/03-OurOwnPriorityQueueImplementation/)

```
top()
  return arr[0]
      insert (x)
        arr.push_back(x);
        while (x is not at the root && x > parent(x))   swap;
 pop()
    Swap arr[0] with arr[arr.size()-1]
    arr.pop_back();
    while(x is not a leaf && x<at least 1 of its children)
            Pick x's largest child and swap
```

Applications of Heaps:

1) Heap Sort: Heap Sort uses Binary Heap to sort an array in O(nLogn) time.
2) Priority Queue: Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations in O(logn) time..
3) Graph Algorithms: The priority queues are especially used in Graph Algorithms.
4) Many problems can be efficiently solved using Heaps. See following for example.
a) K'th Largest Element in an array.
b) Sort an almost sorted array/
c) Merge K Sorted Arrays.

- *Be able to run the insertElement() and removeMax() operations on a heap.*

insertElement() O(h-->log base2 (n+1))
Complexity of the insertion operation is O(h), where **h** is heap's height. Taking into account completeness of the tree, O(h) = O(log n), where **n** is number of elements in a heap.

```
void Heaparr::insert(int da) {
      size++;
      int* tmp = new int[size];

      for (int i = 0; i < size - 1; i++)
            tmp[i] = maxHeap[i];
      tmp[size - 1] = da;
      delete[] maxHeap;
      maxHeap = tmp;
}
```

```
template<int TYPE,typename ITEM>
bool Heap<TYPE,ITEM>::AddItem(ITEM* pItem){
    if (m_iCurrNumOfItems == m_iMaxNumOfItems)
    return false;
    m_aItems[m_iCurrNumOfItems] = pItem;

    for (int i=m_iCurrNumOfItems,j=(i+1)/2-1; j>=0; i=j,j=(i+1)/2-1){
```

```
    if (BestOfTwo(i,j) == i)   SwapItems(i,j);
    else   break;
    }
    m_iCurrNumOfItems++;
    return true;
}
```

Pseudocodes:

```
insert(x)
    arr.push_back(x);
    while(x is not at the root && x>parent(x))  swap

removeMax()    O(h-->log base2 (n+1))
    pop()
        Swap arr[0] with arr[arr.size()-1]
        arr.pop_back();
        while(x is not a leaf && x<at least 1 of its children)
                Pick x's largest child and swap
```

- *Know the running times for all of the priority_queue functions.*
  - Using a vector:
    - push(x)--> call push_back(x)--> O(1) amortized
    - pop() remove max el by shifting and find new max O(n)
    - top() every time you call push(), you check if the newest el is bigger O(1)
  - Using a sorted vector:
    - push(x) needs to maintain a sorted order by shifting O(n)
    - pop() just call pop_back O(1)
    - top() return back() O(1)
  - Using binary heap: (we want)
    - push(x) O(n log n)
    - pop() O(log n)
    - top() O(1)

- **Sorting**
  - *Know how to run at least one of the $O(n^2)$ time sorting algorithms*
    - **bubble sort**

```
void bubblesort(int arr[], int n) {
            isSorted = false;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                isSorted = false;
            }
        }
    }
}
```

- **Insertion sort**

```
void insertionsort(int arr[], int N) {
        for(int i = 1; i < N; i++) {
                int j = i - 1;
                int temp = arr[i];
                while(j >= 0 && temp < arr[j]) {
                        arr[j + 1] = arr[j];
                        j--;;
                }
                arr[j + 1] = temp;
        }
}
```

- Know how to run the subroutines of the O(n log n) time sorting algorithms

- **Merge sort** (you have to know how to run merge())

```
void merge(int *arr1, int n1,int *arr2, int n2, int *result) {
        int *temp = new int [n1 + n2];
        int index1 = 0;
        int index2 = 0;
        int i = 0;
        while(index1< n1 && index2< n2) {
                if(arr1[index1] <= arr2[index2]) {
                        temp[i] = arr1[index1];
                        index1++;
                }
                else {
                   temp[i] = arr2[index2];
                   index2++;
                }
                i++;
        }
        //only runs if arr1 is still full
        while(index1 <n1){
              temp[i] = arr1[index1];
              index1++;
              i++;
        }
        //only runs if arr2 is still full
        while(index2 <n2){
              temp[i] = arr2[index2];
              index2++;
              i++;
        }
        //copy all el from temp into result
        for(int i = 0; i<n1+n2j; i++)
               result[i] = temp[i];
        Delete[temp];
}
```

- **<u>Quicksort</u>** (you have to know how to run partition())

```
int partition (int arr[], int start, int end) {
    int pivotValue = arr [start ];
    int pivotPosition = start;
     for (int pos =start+ 1; pos <= end; pos++){
        if (arr[pos ] < pivotValue ) {
            swap(arr[p i votPosition + 1], arr[pos ] );
            swap(arr[p ivotPosition ], arr[pivotPosition + 1]);
            pivotPosition ++;
        }
    }
    return pivotPosition;
}
    --------------------------------------
void quickSort( int arr[ ] , int start, int end) {
    if (start< end) {
        int p = partition(arr , start, end);
        quickSort (arr, start, p - 1);
        quickSort (arr, p + 1, end);
    }
}
```

<u>Heap sort</u>

```
void heapSort(int arr[], int n)
{
   // Build max heap
   for (int i = n / 2 - 1; i >= 0; i--)
     heapify(arr, n, i);

   // Heap sort
   for (int i=n-1; i>=0; i--)
   {
     swap(arr[0], arr[i]);

     // Heapify root element to get highest element at root again
     heapify(arr, i, 0);
   }
}
```

```
void heapSort(vector<int>& aVector) {

    priority_queue<int> pQueue;

    for(int i=0; i<aVector.size(); i++)
    pQueue.push(aVector[i]);


    for(int i=aVector.size()-1; i>=0; i--) {
    int maximumElement = pQueue.top();
    aVector[i] = maximumElement;
    pQueue.pop();
    }
```

}

Linear search:

```
int searchList (const int  list[], int size, int value) {
        int index= O;
        int position= - 1;
        bool found= false;
       while (index< size && !found) {
              if (list[ index] == value) {
                     found= true;
                     position= index;
              }
              index++;
       }
       return position;
}
```

• *Know the running times for all of the sorting algorithms discussed in class*

sort	Best Case	Worst Case	Average
Insertion Sort	Already sorted, O(n)	Reverse sorted, O(n^2)	O(n^2)
Selection Sort	Already sorted, O(n^2)	Reverse sorted, O(n^2)	O(n^2)
Bubble Sort	Already sorted O(n)	Reverse sorted, O(n^2)	O(n^2)
Merge Sort	O(nlog(n))	Reverse sorted, O(nlog(n))	O(nlog(n))
Quicksort	O(nlog(n))	All elements equal, O(n^2)	O(nlog(n))
Heap Sort	O(n log n)	O(n log n)	O(n log n)

• Sets - contains unique values only!!!

• *Know how to create an empty set.*

```
set<int> first;                         // empty set of ints

int myints[]= {10,20,30,40,50};
set<int> second (myints,myints+5);             // range

set<int> third (second);                   // a copy of second

set<int> fourth (second.begin(), second.end());  // iterator ctor.

set<int,classcomp> fifth;                  // class as Compare

bool(*fn_pt)(int,int) = fncomp;
set<int,bool(*)(int,int)> sixth (fn_pt);  // function pointer as Compare
```

• *Know what the operations insert(), erase(), find(), and count() do, and be able to use them to write a function to solve a problem. (You do **NOT** have to worry about the return type of insert of know how to use it. You must know how to use all other return values)*

pair<iterator,bool> insert(T key); (add key, if not already there) (#include <utility>)

-iterator- el you just added or it's already there

-bool- represents whether or not the insertion was successful (true, if el was just added; false-0, if el was already there)

int erase(T key); (removes key from the set,if it's there; returns wheather el was removed)

```
myset.erase (it);
myset.erase (40);

it = myset.find (60);
myset.erase (it, myset.end());
```

set<T>::iterator find(T el); (returns iterator to el if found, and end() if not found)

```
set<int> myset;
set<int>::iterator it;

// set some initial values:
for (int i=1; i<=5; i++) myset.insert(i*10);// set: 10 20 30 40 50

it=myset.find(20);
myset.erase (it);
myset.erase (myset.find(40));

cout << "myset contains:";
for (it=myset.begin(); it!=myset.end(); ++it)
      cout << ' ' << *it;
cout << '\n';
```

int count(T elem); (like bool returns 1 =, if el is in a set; 0, if el is absent)

```
if (myset.count(i)!=0)
        cout << " is an element of myset.\n";
else
        cout << " is not an element of myset.\n";
```

• *Know how to iterate through a set using set<T>::iterator and const_iterator.*

```
set<int> myset;
set<int>::iterator it;
cout << "myset contains:";
for (it=myset.begin(); it!=myset.end(); ++it)
      cout << ' ' << *it;
```

• Maps

• *Know how to create an empty map.*

```
map<char,int> first;   //empty map

first['a']=10;
first['b']=30;
first['c']=50;
first['d']=70;

map<char,int> second (first.begin(),first.end());

map<char,int> third (second);
```

• *Know what the operations insert(), erase(), find(), count(), and operator[] do, and be able to use them to write a function to solve a problem. (Just as with sets, you need not worry about the return type and value for insert()).*

Built-in functions:

insert(pair<key,value>p);

-returns <iter,bool>   (bool: true/false, depending on if key is there or not)

-inserts the pair into the map

```
map<char,int> mymap;

// first insert function version (single parameter):
mymap.insert (pair<char,int>('a',100) );
mymap.insert (pair<char,int>('z',200) );

pair<map<char,int>::iterator,bool> ret;
ret = mymap.insert ( pair<char,int>('z',500) );
if (ret.second==false) {
        cout << "element 'z' already existed";
        cout << " with a value of " << ret.first->second << '\n';
}

// second insert function version (with hint position):
map<char,int>::iterator it = mymap.begin();
mymap.insert (it, pair<char,int>('b',300));  // max efficiency inserting
mymap.insert (it, pair<char,int>('c',400));  //no max efficiency inserting

// third insert function version (range insertion):
map<char,int> anothermap;
anothermap.insert(mymap.begin(),mymap.find('c'));

// showing contents:
cout << "mymap contains:\n";
for (it=mymap.begin(); it!=mymap.end(); ++it)
        cout << it->first << " => " << it->second << '\n';

cout << "anothermap contains:\n";
for (it=anothermap.begin(); it!=anothermap.end(); ++it)
        std::cout << it->first << " => " << it->second << '\n';
```

int erase(T key); removes the pair from the map whose key is "key"

```
it=mymap.find('b');
mymap.erase (it);                 // erasing by iterator

mymap.erase ('c');                // erasing by key

it=mymap.find ('e');
mymap.erase ( it, mymap.end() ); // erasing by range
```

int count(T key); returns 0 or 1, if the key is in a map or not

```
std::map<char,int> mymap;
char c;

mymap ['a']=101;
mymap ['c']=202;
mymap ['f']=303;

for (c='a'; c<'h'; c++)
{
                std::cout << c;
                if (mymap.count(c)>0)
                std::cout << " is an element of mymap.\n";
                else
                std::cout << " is not an element of mymap.\n";
```

}

find()

```
iterator find (const key_type& k);
const_iterator find (const key_type& k) const;
```

Searches the container for an element with a *key* equivalent to *k* and returns an iterator to it if found, otherwise it returns an iterator to map::end.

```
map<char,int> mymap;
map<char,int>::iterator it;

mymap['a']=50;
mymap['b']=100;
mymap['c']=150;
mymap['d']=200;

it = mymap.find('b');
if (it != mymap.end());
mymap.erase (it);

// print content:
cout << "elements in mymap:" << '\n';
cout << "a => " << mymap.find('a')->second << '\n';
cout << "c => " << mymap.find('c')->second << '\n';
cout << "d => " << mymap.find('d')->second << '\n';
```

- *Know how to iterate through a map using map<T1, T2>::iterator and const_iterator, and know how to handle each element that comes from the iterator (the key is the first component of the pair, and the value is the second component of the pair).* **#include<utility>**

```
for (map<char,int>::iterator it=mymap.begin(); it!=mymap.end(); ++it)
    cout << it->first << " => " << it->second << '\n';
```

```
pair <string,double> product1;                    // default constructor
pair <string,double> product2 ("tomatoes",2.30);   // value init
pair <string,double> product3 (product2);               // copy constructor

product1 = make_pair(string("lightbulbs"),0.99);   // using make_pair (move)

product2.first = "shoes";                         // the type of first is string
product2.second = 39.90;                          // the type of second is double

cout << "The price of " << product1.first << " is $" << product1.second << '\n';
cout << "The price of " << product2.first << " is $" << product2.second << '\n';
cout << "The price of " << product3.first << " is $" << product3.second << '\n';
```

• Binary Search Trees

- *How to create TreeNodes.*

```
struct BSTnode{
    int key;
    BSTnode *left, *right;
}
```

- *Know how to search for an item in a binary search tree.*

```
BSTnode* search (BSTnode* root, int key){
    if (root == NULL) return NULL;
    if (root→ key ==key) return root;
    if (key<root→ key) return search (root→ left, key);
    return search(root→ right, key);
}
```

```
void insert(BSTnode*& root, int key){
        if (root == NULL){
                root == new BSTnode;
                root→ key = key;
                root→ right = root→ left = NULL;
        }
        else if(key<root→ key)
                insert (root→ left, key);
        else insert (root→ right, key);
}
```

• *Know how to visit all nodes in a binary search tree and perform some operation (like printing out all nodes, adding up all nodes, etc.)*

```
void postorder (BSTnode* root){
        if (root ==NULL) return;
        portorder (root→ left);
        postorder (root→ right);
        cout<<root→ key<<endl;
}

void inorder (BSTnode* root){
        if (root ==NULL) return;
        inorder (root→ left);
        cout<<root→ key<<endl;
        inorder (root→ right);
}

void preorder (BSTnode* root){
        if (root ==NULL) return;
        cout<<root→ key<<endl;
        preorder (root→ left);
        preorder (root→ right);
}

void deallocateTree (BSTnode* root){
        if (root ==NULL) return;
        deallocateTree (root→ left);
        deallocateTree (root→ right);
        delete root;
}

int sumAll (BSTnode* root){
        if (root ==NULL) return 0;
        Return sumAll(root-->left) + root→ key + sumAll (root→ right)
}
```

Test questions:

1. From given definition for binary set nodes, write a function that sums all nodes and prints them out
2. Binary max heap→ what happens if you insert an element. Show steps what it looks like before & after
3. Declare a map. Use insert, search.
4. Definitions: binary heap, max heap; binary search tree
5. Write functions using sets and maps

6. Sort array using bubble or insertion sort
7. Run merge on 2 sorted arrays
8. Use sorting on set, map, heap, priority_queue, binary search tree
9. Answer ?: what a running time would be, if we implement sorts with vector/map/queue
10. Know how binary max heap work
11. From a given array, define if it's a binary max heap or not
12. Draw binary max heap, then run insert & delete
13. Run insert & delete on an array and define what an array looks like and what a tree looks like
14. Run a partition algo on an array
15. Use find or count to find an el in a map, don't just use [] in cout
16. Trace through push() and pop() from a given array down the tree (tree push O(log n))
17. Draw a tree from an array→ what happens after using push() and pop()
18. Define if given is a legal max heap
19. Know how to run push() and pop() on binary max heap

Side notes:

```
//adds el to a map
map <string, int> myMap;
myMap.isert(pair<string, int> ("A", 2));
myMap["B"] = 5;
```

```
int kthLargest(const vector <int>& num, int k){
        priority_queue<int> order (num.begin(), num.end());
        for (int i = 0; i<k-1; i++)
                order.pop();
        return order.top();
}
```

In a map insert() returns a pair

Answer to #4:

1. Binary tree is a structure with nodes, where each node has <=2 children
2. A complete binary tree where all levels, except for the last, have to be filled
3. A binary max heap is a complete binary tree, where each node obeys the max heap property for every node x, x<=parent(x) (x=>its children)