CSEC

WEEK 4: DATA STRUCTURES

OCTOBER 21, 2016

COMPUTER SCIENCE ENRICHMENT CLUB

WHAT ARE DATA STRUCTURES?

- A data structure is a specialized format for organizing and storing data.
- We need it when the primitive data types are not enough for our task
- Today we are going to cover-
  - **Static arrays**
  - **Dynamic Arrays**
  - **Queues**
  - **Stacks**
  - **Linked Lists**
  - **Priority Queues**
  - **Sets**
  - **Maps**
  - **Binary Trees**
  - **Heap**

STATIC ARRAYS

- Static array is a data structure, which can store a **fixed-sized** collection of elements of the same data type.

```
int main() {
    // assign all values in one line
    int array1[] = {1,2,3};
    cout << array1[2] << "\n";

    // assign values in multiple lines
    string array2[5];
    array2[0]= "Hi";
    array2[1]= "Bye";
    array2[0]= "Hello";

    cout << array2[0] << "\n";
    cout << array2[3] << "\n";

    return 0;
}
```

```
/*
3
Hello

*/
```

DYNAMIC ARRAYS

- Static array is a data structure, which can store an **undefined** collection of elements of the same data type.

```
int main() {
    vector<int> array1;
    array1.push_back(2);
    array1.push_back(4);
    array1.push_back(6);

    array1[0] = 6;

    cout << array1[0] << "\n";

    cout << array1.size() << "\n";

    array1.push_back(7);

    cout << array1.size() << "\n";

    return 0;
}
```

```
/*
6
3
4
*/
```

QUEUES

- Ordered list of object that follows a first-in-first-out (FIFO) structure.

```
int main() {
    queue <int> q;
    q.push(2);
    q.push(3);
    q.push(4);

    while (!q.empty()){
        cout << q.front() << "\n";
        q.pop();
    }


    return 0;
}
```

```
/*
2
3
4
*/
```

STACKS

- Ordered list of object that follows a last-in-first-out (LIFO) structure.

```
int main() {
    stack <int> s;
    s.push(2);
    s.push(3);
    s.push(4);

    while (!s.empty()){
        cout << s.top() << "\n";
        s.pop();
    }

    return 0;
}
```

```
/*
4
3
2
*/
```

LINKED LISTS

- Ordered list of data each pointing to its successor

```
int main() {
    list<int> my_list;

    my_list.push_back(1);
    my_list.push_back(3);
    my_list.push_back(5);

    for (list<int>::iterator it = my_list.begin(); it != my_list.end(); it++){
        cout << *it << "\n";
    }

    return 0;
}
```

```
/*
1
3
5
*/
```

PRIORITY QUEUES

- An abstract data type like a regular queue, but where additionally each element has a "priority" associated with it. In a priority queue, an element with high priority is served before an element with low priority

```cpp
#include <functional>
#include <queue>
#include <vector>
#include <iostream>

template<typename T> void print_queue(T& q) {
    while(!q.empty()) {
        std::cout << q.top() << " ";
        q.pop();
    }
    std::cout << '\n';
}

int main() {
    std::priority_queue<int> q;

    for(int n : {1,8,5,6,3,4,0,9,7,2})
        q.push(n);

    print_queue(q);

    std::priority_queue<int, std::vector<int>, std::greater<int> > q2;

    for(int n : {1,8,5,6,3,4,0,9,7,2})
        q2.push(n);

    print_queue(q2);

    // Using lambda to compare elements.
    auto cmp = [](int left, int right) { return (left ^ 1) < (right ^ 1);};
    std::priority_queue<int, std::vector<int>, decltype(cmp)> q3(cmp);

    for(int n : {1,8,5,6,3,4,0,9,7,2})
        q3.push(n);

    print_queue(q3);

}
```

SETS

- Abstract data type that can store certain values, in a sorted order, and no repeated values.

```
int main() {
    set <int> my_set;

    my_set.insert(1);
    my_set.insert(2);
    my_set.insert(3);
    my_set.insert(2);
    my_set.insert(3);

    for (set<int>:: iterator it=my_set.begin(); it!=my_set.end(); ++it) {
        cout << *it << "\n";
    }

    return 0;
}
```

```
/*
1
2
3
*/
```

MAPS

- Data type composed of a collection of (key, value) pairs, such that each possible key appears at most once in the collection

```
int main() {
    map<char,int> my_map;

    my_map.insert (pair<char,int>('a',100) );
    my_map.insert (pair<char,int>('z',200) );
    my_map.insert (pair<char,int>('z',500) );

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

```
/*
a => 100
z => 200
*/
```

BINARY TREES

- A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties −
  - The left sub-tree of a node has a key less than or equal to its parent node's key.
  - The right sub-tree of a node has a key greater than or equal to its parent node's key.

HEAP

- A binary heap is a complete binary tree which satisfies the heap ordering property. The ordering can be one of two types:
  - the *min-heap property*: the value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root.
  - the *max-heap property*: the value of each node is less than or equal to the value of its parent, with the maximum-value element at the root.