ALGO-101

Week 4 - Data Structures & Introduction to Graphs and Trees

Rojen Arda Şeşen

ITU ACM

November 2022

Topics

Topics covered at week 4:

- Data Structures
  - Linked List
    - Singly Linked List
    - Doubly Linked List
    - Circular Linked List
  - Stack
  - Queue
  - Deque
  - Priority Queue
- Introduction to Graphs and Trees
  - Graph Representations
  - Tree Representations

Data Structures

Data Structures are ways to store and organize data. Each data structure has its own advantages and disadvantages. It is important to know properties of different data structures to determine which one is more advantageous in different situations.

Linked List

A linked list is a container that stores data in linearly connected nodes. Each node stores its data and a pointer to its associated element, i.e. the next element in the linked list. The first node of the linked list is called the "head" of the linked list and the last element is called the "tail" of the linked list.

Figure: A Singly Linked List

- Linear time ($O(n)$) insertion/deletion.
- Linear time ($O(n)$) random access.

Linked List (cont'd)

A doubly linked list is similar to a generic linked list, with a small difference. It contains another pointer which points to the previous element in the linked list. This pointer is usually called "Prev". This pointer allows us to move back and forth in the linked list as well as the ability to determine the previous element in the linked list in constant time.

Figure: A Doubly Linked List

Linked List (cont'd)

In a circular linked list, the tail of the list is connected to the head of the list hence forming a circle. I.E. "next" of the tail is the head.

Figure: A Circular Linked List

It is convenient (but not necessary) to add dummy nodes at the beginning and the end of the list. These dummy nodes don't store any data but provide a faster way to access first and last items of the list. These are usually called "header" and "trailer".

Sample Singly Linked List Code

```
class LinkedList {
        Node* head;
    public:
        LinkedList() {
            this->head = NULL;
        };
        ~LinkedList();
        bool isEmpty() const;
        Node * getHead() const;
        void addNode(int pos, int
    val);
        void removeNode(int pos);
};
LinkedList::~LinkedList() {
    while (!isEmpty())
        removeNode(0);
}
```

```
class Node {
        int value;
        Node* next;
    public:
        Node(int value) {
            this->value = value;
        };
    friend class LinkedList;
};
bool LinkedList::isEmpty() const {
    return head == NULL;
}

Node* LinkedList::getHead() const{
    return head;
}
```

Sample Singly Linked List Code (cont'd)

```
1  void LinkedList::addNode(int pos,
       int val) {
2      Node* newNode = new Node(val);
3      if (pos == 0) {
4          newNode->next = head;
5          head = newNode;
6          return;
7      }
8      Node * cursor = head;
9      while (pos > 1 && cursor->next
       ) {
10         cursor = cursor->next;
11         pos--;
12     }
13     newNode->next = cursor->next;
14     cursor->next = newNode;
15 }
```

```
1  void LinkedList::removeNode(int
       pos) {
2      if (isEmpty()) return;
3      Node * temp = head;
4      if (pos == 0)
5          head = head->next;
6      else {
7          Node * cursor = head;
8          while(pos > 1 && cursor) {
9              cursor = cursor->next;
10             pos--;
11         }
12         temp = cursor->next;
13         cursor->next = temp->next;
14     }
15     delete temp;
16 }
```

Stack

A stack is a container that stores data according to **LIFO (last-in first-out)** principle.

- **top**: last added element of the stack.
- **push**: inserting an element at the top of the stack.
- **pop**: removing an element from the top of the stack.

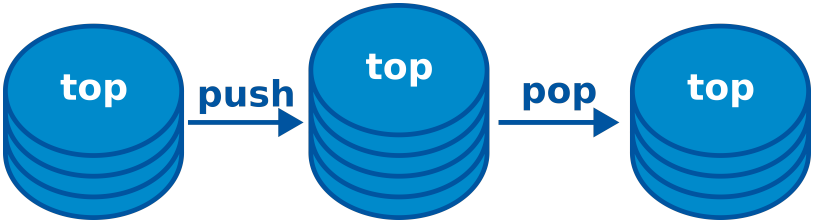


Figure: A Stack

Stack (cont'd)

LIFO means that the last element that is *pushed* to the stack will be first one removed. Stack data structure does not offer random access. Only the *top* element is accessible at any time. Push and Pop (insertion and deletion) operations take constant ($O(1)$) time. Stacks are extremely useful for *backtracking*.

Operation	**Time**
size	$O(1)$
isEmpty	$O(1)$
top	$O(1)$
pop	$O(1)$
push	$O(1)$

Table: Stack Operations' Time Complexity

Sample Stack Code

```
1  class Stack {
2  private:
3      Node * items;
4      int itemCount;
5  public:
6      Stack();
7      bool isEmpty() const;
8      int top() const;
9      void pop();
10     void push(int value);
11     int size() const;
12 };
```

```
1  Stack::Stack() {
2      items = NULL; itemCount = 0;
3  }
4  void Stack::push(int value) {
5      Node* newNode = new Node(value
       );
6      newNode->next = this->items;
7      this->items = newNode;
8      itemCount++;
9  };
10
11 int Stack::size() const {
12     return itemCount;
13 };
```

Note that we've utilised linked list when implementing a stack.

Sample Stack Code (cont'd)

```
1  bool Stack::isEmpty() const {
2      return items == NULL ? true : false;
3  }
4
5  int Stack::top() const {
6      return isEmpty()? -1 : items->value;
7  }
8
9  void Stack::pop() {
10     if (isEmpty())
11         cout << "Stack is empty." << endl;
12     else {
13         items = items->next;
14         itemCount--;
15     }
16 }
```

C++ STL Stack

An implementation of stack data structure is present in the Standard Template Library, under "**stack**" header. The implementation is based on STL Vector class.

Stack Initialization

```cpp
#include <stack>

int main() {
    std::stack<int> s;
    s.push(5);
    s.pop();
}
```

Some STL stack Operations

push(x)	Adds x to the stack.
top()	Returns the top element of the stack.
pop()	Removes the top element from the stack.
empty()	Returns true if the stack is empty, false otherwise.
size()	Returns the number of elements in the stack.

Table: Stack Operations

All of these operations take $O(1)$ time.

Queue

Queue data structure is a container that stores data according to the **FIFO (first-in first-out)** principle.

- **front**: First element to be removed from the queue.
- **push**: Inserting an element at the back of the queue, i.e. *enqueue*.
- **pop**: Removing an element from the front of the queue, i.e. *dequeue*.

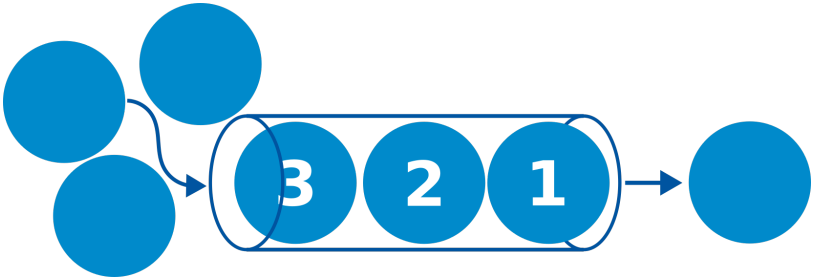


Figure: A Queue

Properties of Queues

- Just like stacks, queues don't allow random access. Only the front and the back of the queue (first pushed element) is accessible at any time.
- The elements enter the queue from the **back** and are removed from the **front**.
- All queue operations take constant ($O(1)$) time.

You can think of the queue as a line of people waiting in the cafeteria.

Figure: A Line of People

Sample Queue Code

We will use a doubly linked list to implement the queue.

```
class Queue {
        Node * front;
        Node * back;
        int size;
    public:
        Queue();
        void push(int x);
        void pop();
        int getFront() const;
        bool isEmpty() const;
        int getSize() const;
};
```

```
Queue::Queue() {
    front == NULL;
    back == NULL;
    size = 0;
};
void Queue::pop() {
    Node * temp = front;
    front = front->prev;
    delete temp;
    size--;
};
```

Sample Queue Code (cont'd)

```
1  void Queue::push(int x) {
2      Node * newItem = new Node(x);
3      if (isEmpty()) {
4          front = newItem;
5          back = newItem;
6      }
7      else {
8          newItem->next = back;
9          back->prev = newItem;
10         back = newItem;
11     }
12     size++;
13 };
```

```
1  int Queue::getFront() const {
2      return front->value;
3  };
4
5  bool Queue::isEmpty() const {
6      return front == NULL;
7  };
8
9  int Queue::getSize() const {
10     return size;
11 };
```

C++ STL Queue

An implementation of queue data structure is present in the Standard Template Library, under "**queue**" header. The implementation is based on STL Vector class.

Stack Initialization

```
#include <queue>
using namespace std;

int main() {
    queue<int> q;
    q.push(5);
    cout << q.front();
    q.pop();
}
```

Some STL stack Operations

push(x)	Adds x to the back of the queue.
front()	Returns the front element of the queue.
back()	Returns the back element of the queue.
pop()	Removes the front element from the queue.
empty()	Returns true if the queue is empty.
size()	Returns the number of elements in the queue.

Table: Queue Operations

All of these operations take $O(1)$ time.

Double-Ended Queue (Deque)

Double-ended Queue is a queue like data structure but it supports insertion and deletion at both front and the end of the queue, hence the name "Double-Ended Queue". This data structure is usually called "deque".

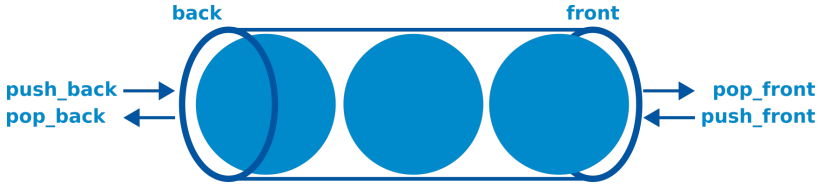


Figure: A Deque

Sample Deque Code

We will use a doubly linked list to implement the deque.

```
class Deque {
    Node * front;
    Node * back;
    int size;
    public:
        Deque();
        void pushFront(int x);
        void pushBack(int x);
        void popFront();
        void popBack();
        int getFront() const;
        int getBack() const;
        int getSize() const;
        bool isEmpty() const;
};
```

```
Deque::Deque() {
    front = NULL;
    back = NULL;
    size = 0;
};
```

Sample Deque Code (cont'd)

```
1  void Deque::pushFront(int x) {
2      Node* newNode = new Node(x);
3      if (isEmpty()) {
4          front = newNode;
5          back = newNode;
6      }
7      else {
8          front->next = newNode;
9          newNode->prev = front;
10         front = newNode;
11     }
12     size++;
13 };
```

```
1  void Deque::pushBack(int x) {
2      Node* newNode = new Node(x);
3      if (isEmpty()) {
4          front = newNode;
5          back = newNode;
6      }
7      else {
8          back->prev = newNode;
9          newNode->next = front;
10         back = newNode;
11     }
12     size++;
13 };
```

Sample Deque Code (cont'd)

```
1  void Deque::popFront() {
2      Node * temp = front;
3      front = front->prev;
4      front->next = NULL;
5      delete temp;
6      size--;
7  };
8
9  void Deque::popBack() {
10     Node * temp = back;
11     back = back->next;
12     back->prev = NULL;
13     delete temp;
14     size--;
15 };
```

```
1  int Deque::getFront() const {
2      return front->value;
3  };
4
5  int Deque::getBack() const {
6      return back->value;
7  };
8
9  int Deque::getSize() const {
10     return size;
11 };
12
13 bool Deque::isEmpty() const {
14     return back == NULL;
15 };
```

C++ STL Deque

An implementation of deque data structure is present in the Standard Template Library, under "**queue**" header. The implementation is based on STL Vector class. Unlike std::stack and std::queue, std::deque supports insertion and random access.

Code

```cpp
#include <deque>
using namespace std;

int main() {
    deque<int> dq;
    dq.push_back(2);              // [2]
    dq.push_front(3);             // [3, 2]
    dq.push_back(1);              // [3, 2, 1]
    cout << dq[1] << endl;        // >>> 2
    cout << dq.back() << endl;    // >>> 1
    cout << dq.front() << endl;   // >>> 3
    dq.pop_back();                // [3, 2]
    dq.pop_front();               // [2]
    dq.insert(dq.end(), 3);       // [2, 3]
    dq[0] = 1;
}
```

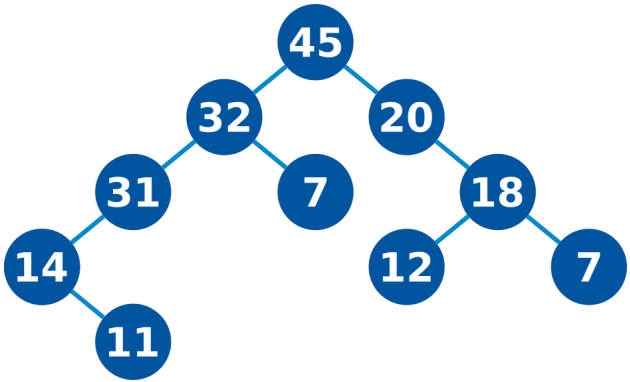
Deque Operations

push_back(x)	Adds x to the back of the deque.
push_front(x)	Adds x to the front of the deque.
front()	Returns the front element of the deque.
back()	Returns the back element of the deque.
pop_back()	Removes the back element from the deque.
pop_front()	Removes the front element from the deque.
empty()	Returns true if the queue is empty.
size()	Returns the number of elements in the queue.
insert(iterator, value)	Inserts the given value to the given index.
erase(iterator)	Erases the element in the given index.

Table: Deque Operations

Priority Queue

Priority queue is an important data structure for storing elements in order, based on their priority. The element with the highest priority can be accessed and removed. Implementation of the priority queue is based on heaps. Heaps and priority queue implementation will not be covered in this lecture.



Right Part text

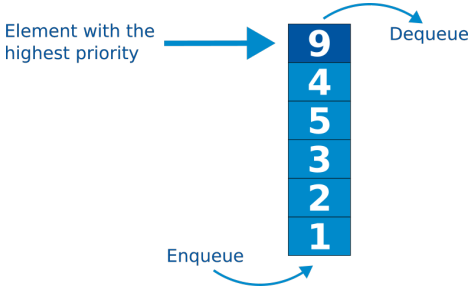


Figure: Priority Queue

Priority Queue (cont'd)

```cpp
#include <iostream>
#include <queue>
using namespace std;

int main() {
    priority_queue<int> pq;
    pq.push(1);
    pq.push(12);
    pq.push(2);
    pq.push(98);
    pq.push(6);
    pq.push(3);
    while (!pq.empty()) {
        cout << pq.top() << " ";
        pq.pop();
    }
}
```

Elements are placed in their rightful position as they're inserted. Insertion takes O($logn$) time. Inserting n elements takes O($nlogn$) time. STL priority queue is defined in the header queue .

Output:
98 12 6 3 2 1

Priority Queue (cont'd)

A "greater" function could be used to sort numbers in ascending order in priority queue, i.e. a max heap.

```
#include <iostream>
#include <queue>
using namespace std;

int main() {
    priority_queue <int, vector<int>, greater<int> > pq;
    pq.push(1);
    pq.push(12);
    pq.push(2);
    pq.push(98);
    while (!pq.empty()) {
        cout << pq.top() << " ";
        pq.pop();
    }
}
```

Output: 1 2 12 98

Priority Queue With Custom Comparators

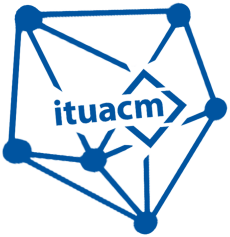
It is also possible to write your own comparator functions by overloading () operator.

```
struct Comparator {
    bool operator() (const pair<int, int>& a, const pair<int, int>& b) {
        return a.second < b.second;
    }
};

int main() {
    priority_queue<pair<int, int>, vector<pair<int,int>>, Comparator> pq;
    pq.push({5, 2});
    pq.push({99, -1});
    pq.push({3, 4});
    pq.push({6, 1});
    while (!pq.empty()) {
        cout << pq.top().first << ", " << pq.top().second  << endl;
        pq.pop();
    }
}
```

Graphs

A graph is a set of **nodes**, some of which are connected with **edges**.

- **node:** A node or a **vertex** is a structure that contains data.
- **edge:** An edge is the connection between two nodes, they are also called **arcs**. Edges can be
  - directed/undirected
  - weighted/unweighted



Graphs (cont'd)

- Two nodes are said to be **adjacent** if there exists an edge between them. A graph node can be adjacent to zero or more nodes.
- A graph is **connected** if there is a path from any node to any other node in the graph.

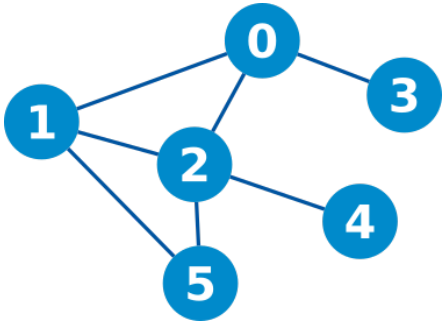


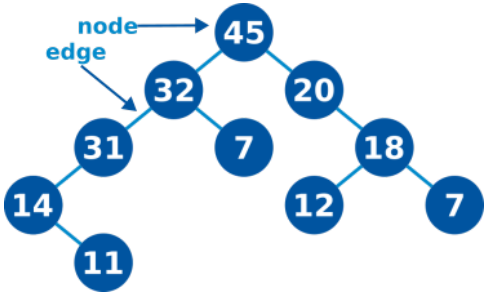
Figure: An undirected, unweighted graph.

Trees

Tree is a nonlinear data structure that stores data hierarchically. A tree is made of **nodes** and **edges**. Trees are specialized types of graphs.

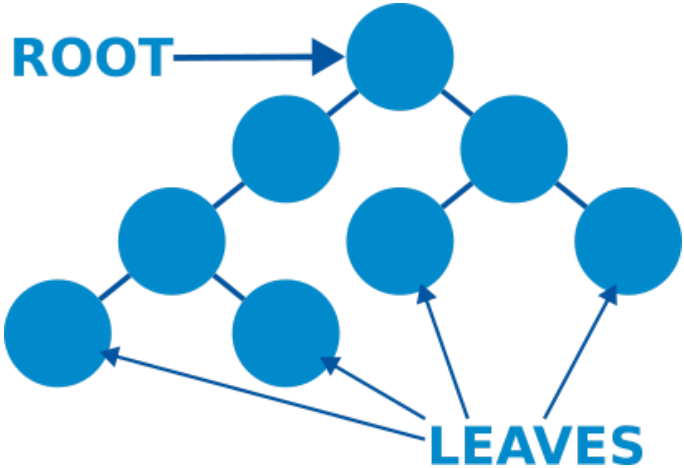
- **node:** A tree node is where the data of the tree is stored. It's also called a **vertex**.
- **edge:** A tree edge is the connection between two tree nodes. Edges can be weighted and unweighted.

Each node has zero or more **children** and except the first node of the tree, **root**, each node has exactly one **parent**.



Trees (cont'd)

- The first node of the tree is called the tree's **root**.
- Outer nodes of the tree are called the **leaves** of the tree.



Trees (cont'd)

- Children of the same parent are called **siblings**.
- Node a is **ancestor** of node b if a $=$ b or a is an ancestor of b's parent.
- Node b is **descendant** of node a if a is an ancestor of b.
- A **subtree** with root v is all descendent nodes of v.

Graph and Tree Representations

There are many ways to represent graphs and trees in computers. We will be covering:

- Node based implementations
- Adjacency List
- Adjacency Matrix

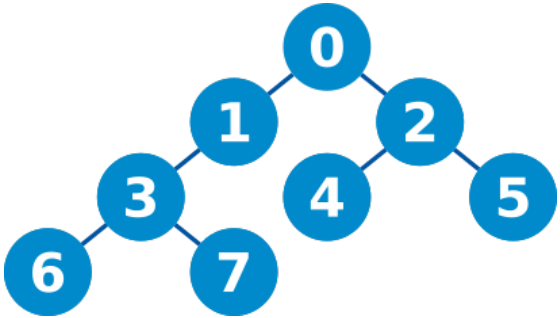
C++ Node Based Implementations

```cpp
class GraphNode {
public:
    int data;
    vector<GraphNode*> adj;
};

class Graph {
public:
    vector<GraphNode*> nodes;
    /*
    GraphNode* startingNode;
    */
};
```

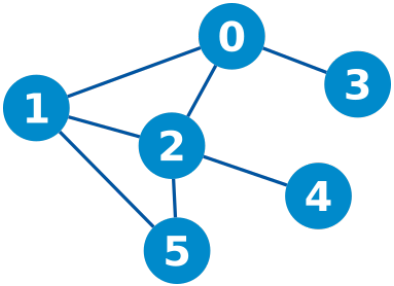
```cpp
class TreeNode {
public:
    int data;
    vector<TreeNode*> children;
};

class Tree {
public:
    TreeNode* root;
};
```

Adjacency List

For each node, we keep a list of its adjacent nodes.



Adj. List: [[1, 2], [3], [4, 5], [6, 7], [], [], [], []]



Adj. List:
[[1, 2, 3], [0, 2, 5], [0, 1, 4, 5], [0], [2], [1, 2]]

Adjacency Matrix

We will create a matrix sized n x n where n is the number of nodes in the graph/tree we want to represent. We place a "1" if there is an edge between nodes, 0 otherwise.

	0	1	2	3	4	5	6	7
0	0	1	1	0	0	0	0	0
1	0	0	0	1	0	0	0	0
2	0	0	0	0	1	1	0	0
3	0	0	0	0	0	0	1	1
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0

Table: Adjacency Matrix of the tree

	0	1	2	3	4	5
0	0	1	1	1	0	0
1	1	0	0	0	0	1
2	1	0	0	0	1	1
3	1	0	0	0	0	0
4	0	1	0	0	0	0
5	1	1	0	0	0	0

Table: Adjacency Matrix of the graph

References

Michael T. Goodrich, Roberto Tamassia, David M. Mount - Data Structures and Algorithms in C++ 2nd Edition - Wiley (2011)

Reviewer: Novruz Amirov