DSA NOTES

Linked List

Node \*firstNode = NULL;

Node \*temp Node = new Node;

Temp->item = “ “;

Temp->next = NULL;

If ( firstNode == NULL){

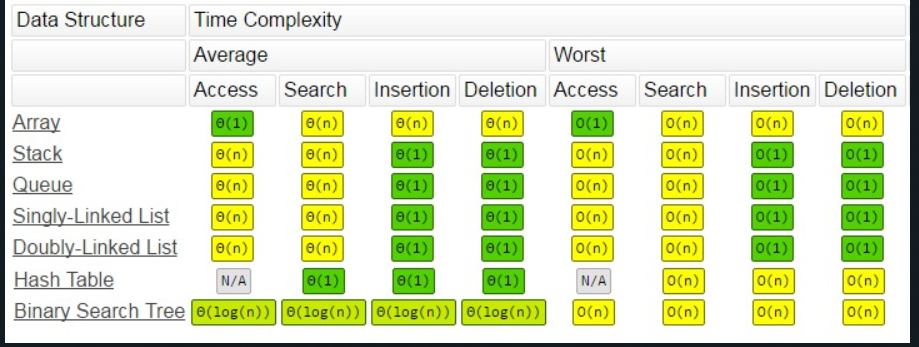
firstNode = temp;

} else {

firstNode->next = temp;

}

Big O Notation



Stack

Last-In-First-Out (LIFO)

* Language Processing
* Balancing of symbols
* Expression conversion (infix to postfix, postfix to prefix, etc)
* Reversing
* “undo” mechanism in text editors
* Reversing a string
* Backtracking
* Find way through a maze
* Find path from one point to another point

Array based:

Stack() // constructor

~Stack() //deconstructor

Bool : Push(ItemType item) // add

{

Bool Success = top < MAX\_SIZE -1;

If (success){

Top++;

Item[top] = item;

}

Return success;

}

Pop() // remove

getTop()

isEmpty()

Pointer based:

Stack() // constructor

~Stack() //deconstructor

Push() // add

Pop() // remove

getTop()

isEmpty()

Queue

First-In-First-Out (FIFO)

Lecture5 - Slide 14 and 16 Enqueue and Dequeue diagram

Pointer Base:

Queue.h

Class Queue {

private:

struct Node{

ItemType item;

Node \*next;

};

Node \*frontNode;

Node \*backNode;

public:

Queue();

~Queue();

bool enqueue(ItemType&); // add

bool dequeue(); //remove

bool dequeue(ItemType&); // remove from index

}

Array Base: (Circular Queue)

*// Queue.h - Specification of Queue ADT (implemented using Circular Array)*

#pragma once

#include<string>

#include<iostream>

using namespace std;

typedef int ItemType;

class Queue

{

private:

ItemType items[MAX\_SIZE];

int front;

int back;

bool isFull;

public:

// constructor

Queue();

*// enqueue (add) item at back of the queue*

bool enqueue(ItemType item);

*// dequeue (remove) item from front of the queue*

bool dequeue();

*// retrieve (get) and dequeue item from front of the queue*

bool dequeue(ItemType& item);

*// retrieve (get) item from front of queue*

void getFront(ItemType& item);

*// check if the queue is empty*

bool isEmpty();

};

1. Queue();

Queue::Queue(){

front = 0;

back = MAX\_SIZE -1;

isFull = false;

}

1. bool enqueue(ItemType item);

bool Queue::enqueue(ItemType item){

if (!isFull){

if (back+1%MAX\_SIZE == front)){

isFull = true;

}

back = (back+1) % MAX\_SIZE;

items[back] = item;

return true;

}

Else{

cout << “Queue full ,cannot enqueue” << endl;

return false;

}

}

1. bool dequeue(ItemType& item);

bool Queue::dequeue(ItemType& Item){

if (!isEmpty()){

item = items[front]

front = (front+1) % MAX\_SIZE;

isFull = false;

return true;

} else {

cout << “Queue Empty, Cannot dequeue“ << endl;

return false;

}

}

1. bool isEmpty();

bool Queue::isEmpty(){

return (!isFull() && (back+1)% MAX\_SIZE == front)  
}

Hash Table

// Dictionary.h - - Specification of Dictionary ADT

#include<string>

#include<iostream>

using namespace std;

const int MAX\_SIZE = 100;

typedef string ItemType;

typedef int KeyType;

struct Node

{

KeyType key; // search key

ItemType item; // data item

Node \*next; // pointer pointing to next item

};

class Dictionary

{

private:

Node \*items[MAX\_SIZE];

int size; // number of items in the Dictionary

public:

// constructor

Dictionary();

// destructor

~Dictionary();

int hash(KeyType key);

// add a new item with the specified key to the Dictionary

bool add(KeyType newKey, ItemType newItem);

// remove an item with the specified key in the Dictionary

void remove(KeyType key);

// get an item with the specified key in the Dictionary (retrieve)

ItemType get(KeyType key);

// check if a specified key is in the Dictionary

bool contains(KeyType key);

// check if the Dictionary is empty

bool isEmpty();

// check the size of the Dictionary

int getLength();

//------------------- Other useful functions -----------------

// display the items in the Dictionary

void print();

};

Dictionary.cpp

int Dictionary::hash(KeyType key){

Return key % MAX\_SIZE;

}

bool Dictionary::add(KeyType newKey, ItemType newItem){

int index = hash(newKey);

if (items[index] == NULL) {

items[index] = new Node;

items[index]->key = newKey;

items[index]->item = newItem;

items[index]->next = NULL;

}

else {

Node\* current = items[index];

if (current->key != newKey) { return false; }

while (current->next != NULL) {

current = current->next;

if (current->key == newKey) { //duplicate data

return false;

}

}

Node\* temp = new Node;

temp->item = newItem;

temp->key = newKey;

temp->next = NULL;

current->next = temp;

}

size++;

return true;

}

Bool Dictionary::remove(KeyType key){

int index = hash(key);

if (items[index] != NULL) {

Node \*current = items[index];

if (current->key == key){

items[index] == items[index]->next;

delete current;

size--;

return true;

} else {

while (current->next != NULL) &&

(current->next->key != key) {

current = current->next;

}

if (current->next != NULL){

Node \*temp = current->next;

current->next = temp->next;

delete temp;

temp = NULL;

size--;

return true;

}

}

}

return false;

}