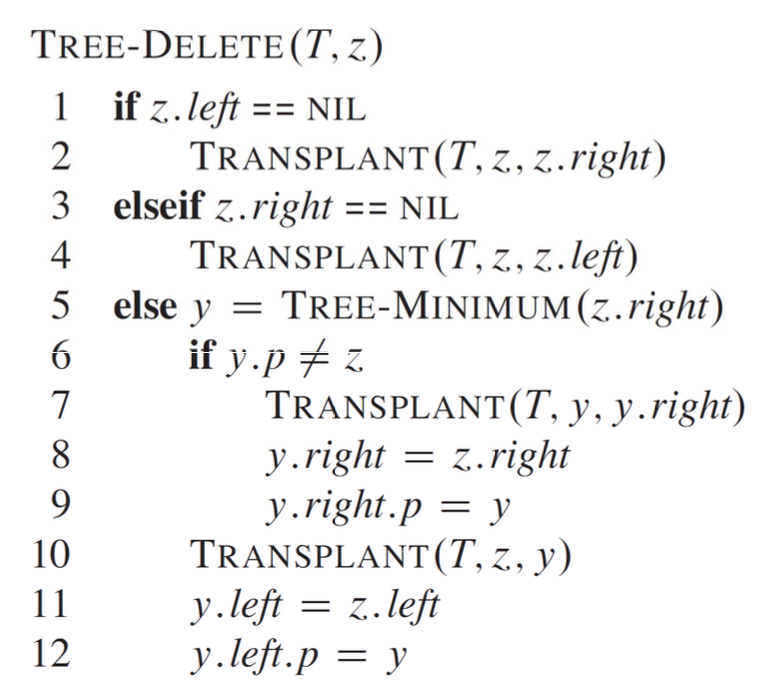
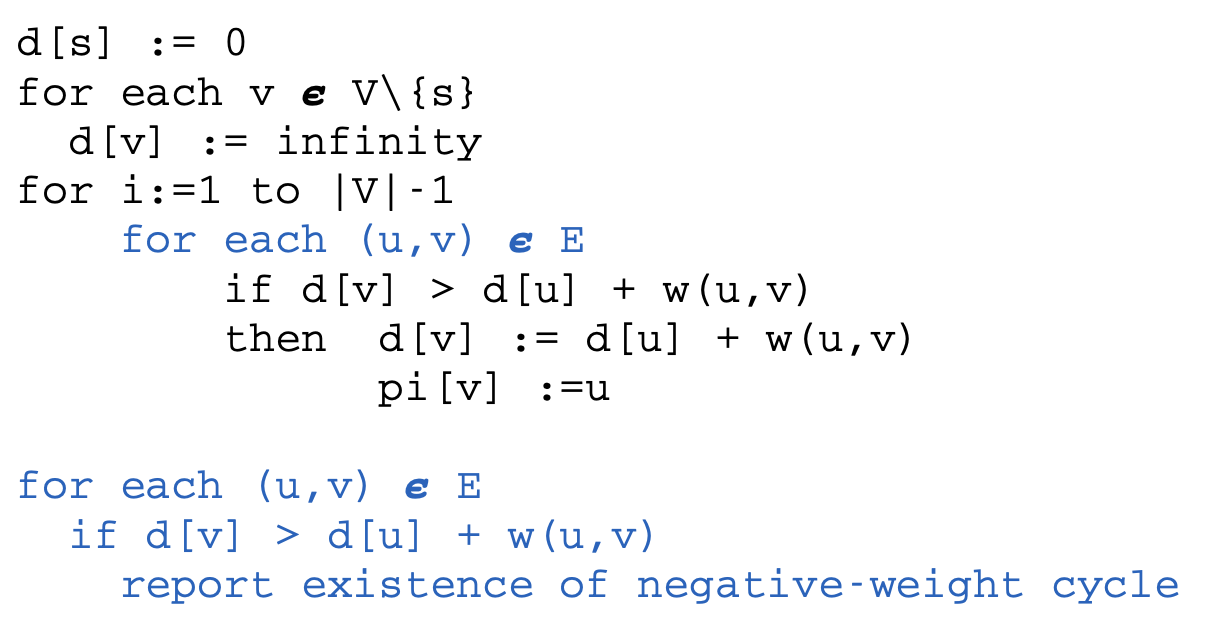
BELLMAN-FORD

DIJKSTRA



ADS





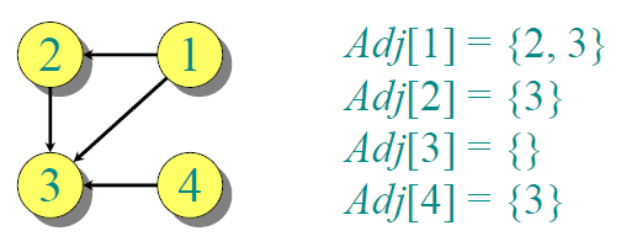
DFS. O(V+E)

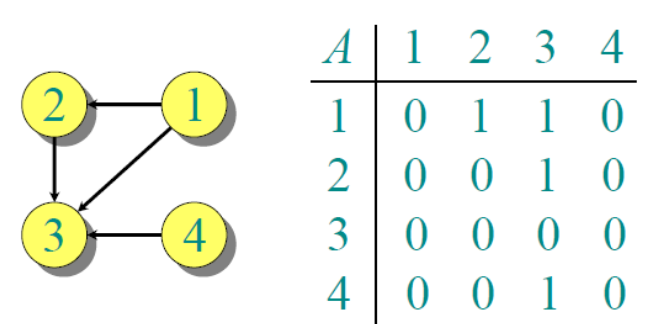
BFS. O(V+E)

PRIMS. O(V^2)with array, O(ElogV) with min-heap

DIJKSTRA (V^2) with array, O(ElogV) with min-heap

BELLMAN-FORD (V\*E)

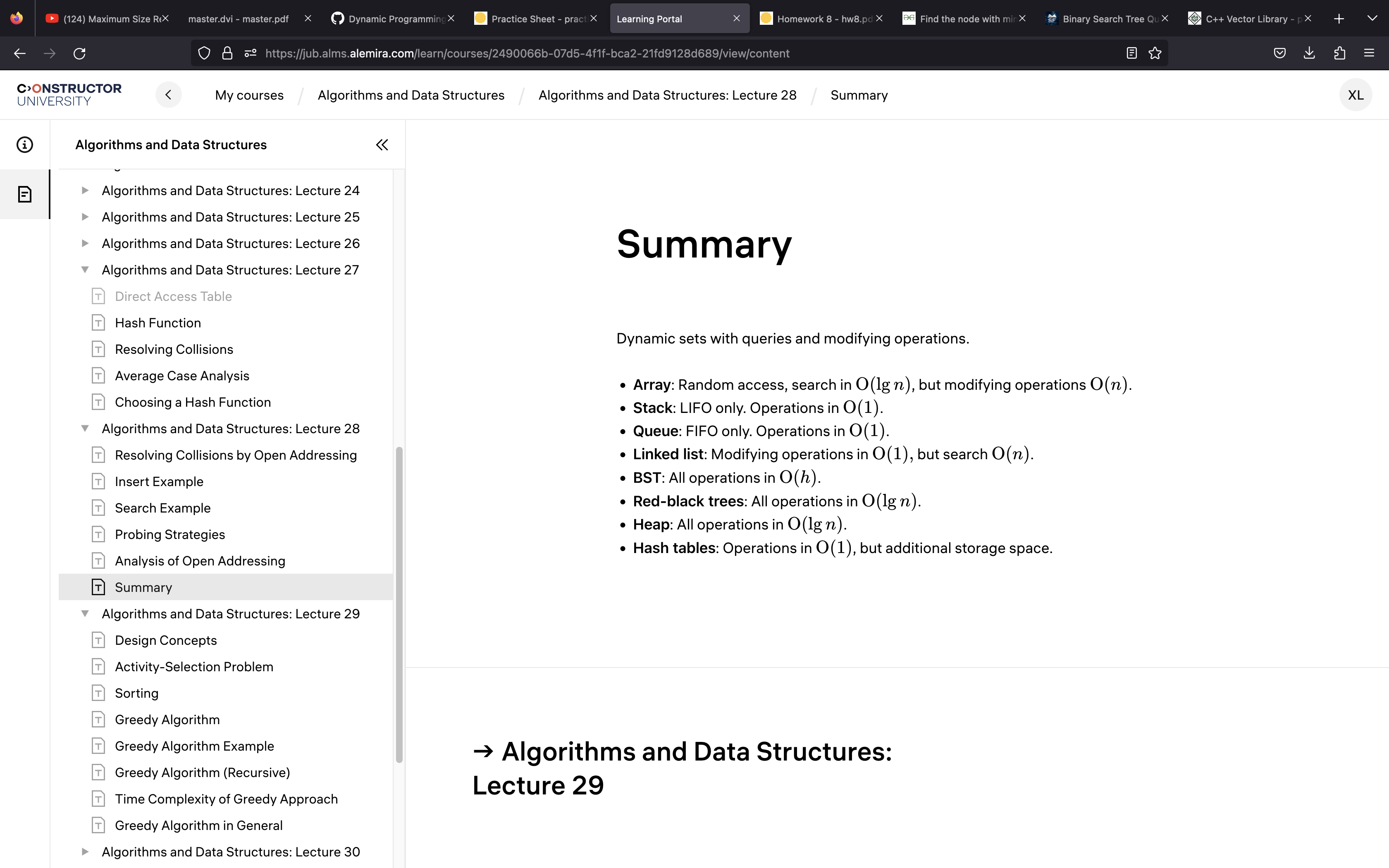
Adj list

Adj matrix

Greedy approach:

O(n) if sorted

O(nlogn) if not sorted



Probing strategies:

-linear probing the below is used both for insert and search

h(x)=x%size of table

if there’s a collision do:

h’(x)=(h(x)+f(i))% size of table =>f(i)=i

i=0,1,2,3,4,5, ex(inset 3,13, table=10)3%10=3, than 13%10=3, but taken, do (13+1)%10=4

-quadratic probing

h(x)=x%size of table

if there’s a collision do:

h’(x)=(h(x)+c1i+c2\*i^2)% size of table

-double hashing

H(I,k)=(h1+ih2)%m

Postorder traversal: left,right,parent

public void func(Tree root)

{

func(root.left());

func(root.right());

System.out.println(root.data());

}. Preorder traversal: parent left,right,

public void func(Tree root)

{

System.out.println(root.data());

func(root.left());

func(root.right());

}

RED-BLACK TREE: (modifying operations in O(lgn))

All leaves have nill as key

1-Either node is red pr black

2-The root is black

3-all leaves(nill) are black

4-If a node is red than both children are black

5-For each node all paths from root to leaf have the same nr. of black nodes

\*\*\*don’t rotate immediately see if you can change colors to maintain the rbt property and than rotate possible a subtree not the whole

BST:

-inorder visit(T):

If(t--nil)return;

Inorder visit(t.left);

Operation();

Inordervisit(t.right);

-search

-minimum. O(h)

-maximum

-sucessor

-predecessor. O(h)

Insert O(h)

-transplant O(1)\*\*-delete O(h)

S

Bst insert(T,x)

Y=t.root;

If(y==nil)

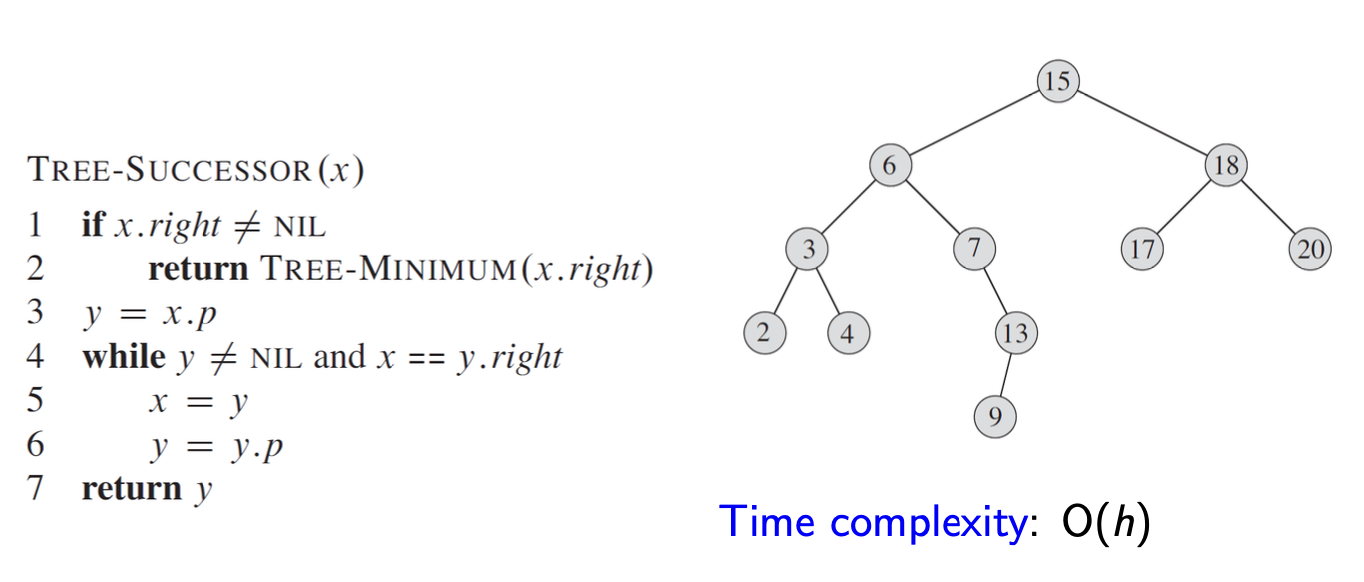
t.root=Newnode(x)

If(x>y.key)

y.right=Insert(y.right,x);

If(x<y.key)

y.left=Insert(y.left,x)



Tree-predecessor(x)

If(x.left!=nil)

Return tree\_max(x.left)

Y=x.p

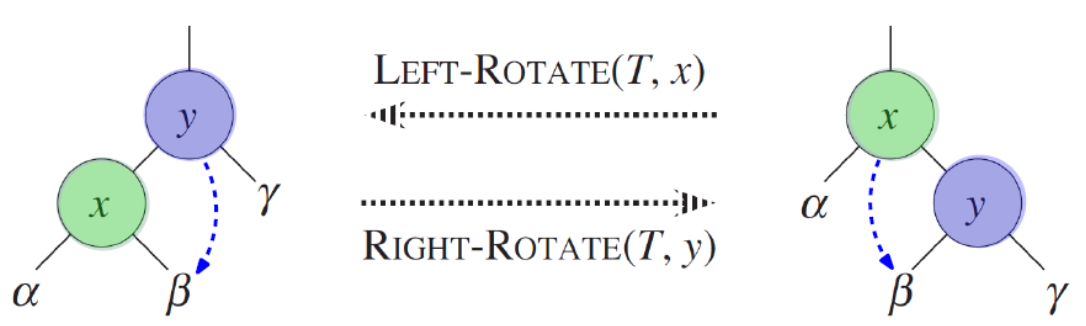
While(y!=nil && x=y.left)

X=y;

Y=y.p

return y;

ADS



Asymptotic analysis:

-f(n)=Θ(g(n)) a=b)

-f(n)=O(g(n)). a<=b

-f(n)= Ω(g(n)) a>=b

-f(n)=o(g(n)) a<b

-f(n)= ω(g(n)) a>b

DATA STRUCTURE: a way to organize data to facilitate access

-array

-heap

-linked list

-max priority queue

-stack

-queue

-linked list

Sorting algorithms:

-insertionsort O(n^2), insitu

-meregesort O(nlogn), non-insitu

-heapsort(max,extract-max,increase key,min,insert) O(nlogn), insitu

-quicksort O(nlogn),insitu

-countingsort O(n+k)

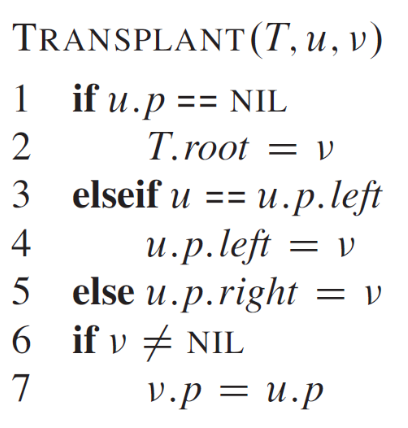
-radixsort

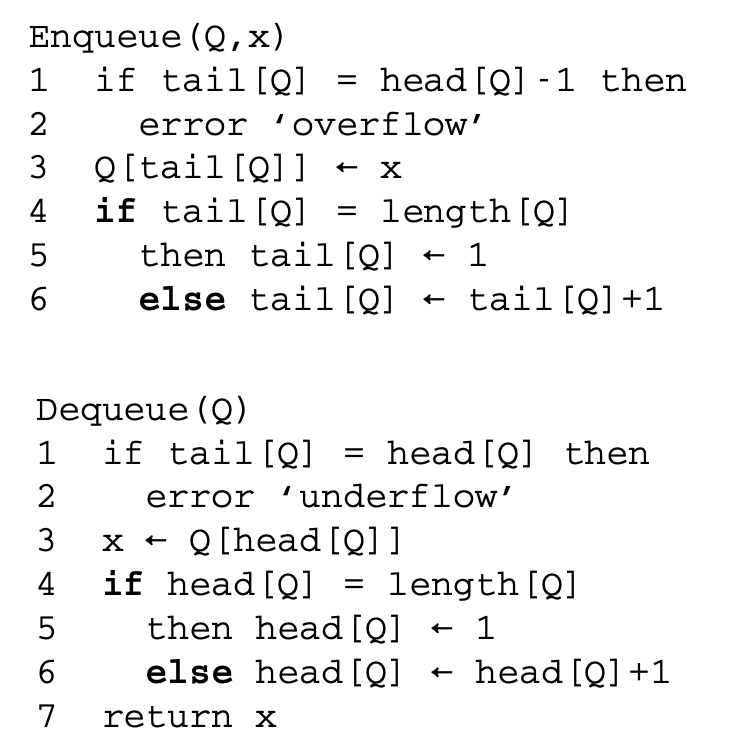
-bucketsort O(n^2),O(n)

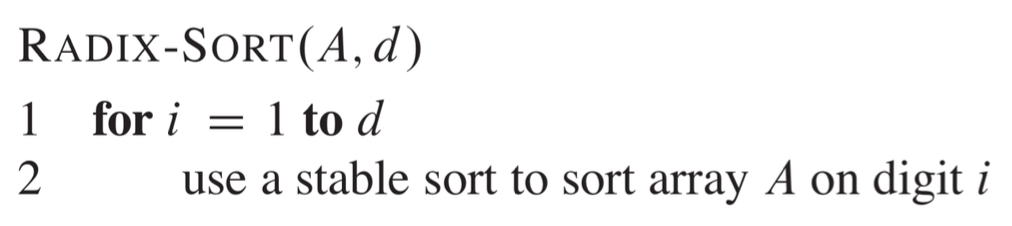
-radixsort O(nd);

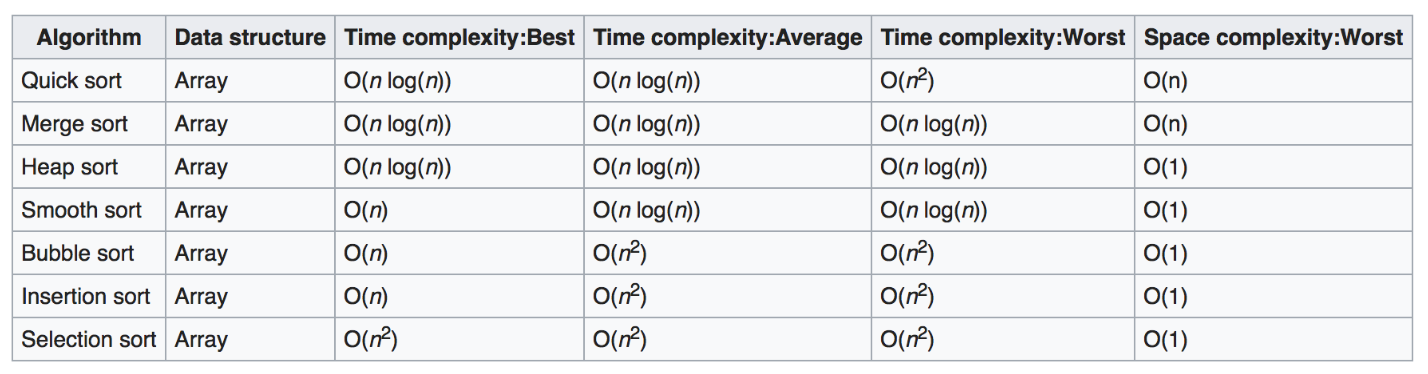
Types of iterators: input,output,forward,bidirectional,

random access









Loop invariant proof:

-Initialisation (proof holds before the first loop:ex one element in the data structure)

-maintanance (each iter. Maintains the loop invariant)

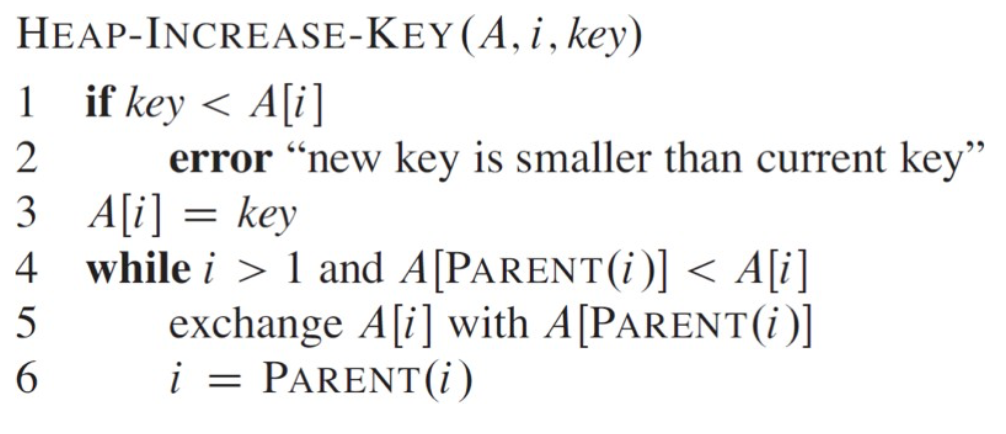
-termination (look what happenes when the loop terminates)

Master method

Substitution

Recursion tree





-Common to all containers:

Begin(), End(), Erase(), Size()

-Specific:

Associative: insert(const value\_type &x)

Sequence: insert(interator p,const value\_type&x)

Algorithms: process elemets of containers like search,sort,modify

Containers: manage collections of data

-Associative, work with pairs of key,value ,sorted efficient in accessing els by their key: set,map,multimap,multiset

-Sequence (position depends on place of insertion) efficient in accesing els by their position: vector, deque, list

Iterators: navigate through the elelements of the container

**Operations**

* \* returns element at current position
* ++ step forward to next element
* == equals same position
* != not equals same position

RED-BLACK trees:24,25 O(logn)

-5 properties

-rotation

RRotate(t,y)

X=y.left

Y,r=x.l

If(x.left!=t.nil)

x.left.p=y

y,p=x.p

If(y.p==nil)

X=t.root

Elseif(y=y.p.left)

y.p.left=x

else

y.p.right=x

-insert

-transplant

Containers:

Set: insert,erase,count,find,size,empty,clear,begin,end,rbegin, rend,lower\_bound(),upper\_bound(),equal\_range()

Multiset: insert,erase,count,find,size,empty,clear,begin,end,rbegin, rend,lower\_bound(),upper\_bound(),equal\_range(), intersection, union , symmetric difference set\_union(a.begin(),a.end(),  b.begin(), b.end(), 10 inserter(c, c.begin()));

Map: insert,erase,count,find,at,operator,size,empty,clear,begin,end,rbegin,rend,lower\_bound,upper\_bound,equal\_range

Multimap: insert,find,erase,count,size,empty,clear,begin,end,rend,rbegin,lower\_bound, upper\_bound(),equal\_range

Vector: push\_back,pop\_back,resize,reverse,at,operator[],front,back,clear,empty,size,erase,insert,swap,begin,end,find,replace

List: push\_back,push\_front,insert,erase,clear,empty,size,front,back,reverse,sort,splice,merge,unique,begin,end,find

Deque: std::deque supports all the operations of std::vector, as well as some additional operations, such as push\_front(), pop\_front(), emplace\_front(), front(), and back(), erase

Queue: push,pop,front,back,empty,size, NO ERASE, NO INSERT , NO FIND\*\*\*

Stack: push, pop, size, top, empty \*\*\*NO ERASE, NO FIND

std::stack is implemented as an adapter on top of another container, usually std::deque, std::list, or std::vector, and provides only a limited set of operations. Therefore, you cannot access or modify elements in the middle of the stack, nor can you iterate over the elements in the stack.

Sentinels:21:

Search(L,s):

X=next[nil[L]];

While(x!=nil[l] && x,value!=s){

X=x->next;

}

Return x;

-insert(L,s){

s->next=next[nil[l]];

next[nil[l]]->prev=s;

next[nil[l]]=s;

s->prev=nil[l];

return L;

}

-Delete(L,x){

next[prev[x]]=nexy[x];

prev[next[x]]=prev[x];

}

D-ary trees:

struct node{

int val;

node\*parent;

node\*child[n];

};

Data structure Operation complexity

Heap insert,delete logn

Search, heapify. N

Sorting,creating a heap nlogn

Max-heap. Insert,delete log n

Max O(1)

Search O(n)

Queue enque,deque,peek(not remove). O(1)

BST insert,delete,search. O(h)

Lists(depends, single,linked, doubly linked)

Binary tree: 23

-inorder(left,operat,right)

-treemax,treemin,successor,predecessor(\*when left/right==nill)

-insert, transplant

-deletion Tree search(x,k)

If(x==nill || k==x.k)

Return x

If(k<x.key)

Return treeserach(x,x.left)

Return treeseach(x,x.right)

Struct node{

Int val;

Node \*left;

Node\* right;

Node\*parent;

};

The containers which don’t have the [] oper. Overloaded:

set,list,stack,queue,multiset

Queues:

Lists:

Insertion=deletion at the middle/end ->O(1), if we are given the pointer not the key

template<class T>

void Stack<T>::extend(){

T \*newptr=new T[2\*size];

for(int i=0;i<size;i++){

newptr[i]=ptr[i];

}

delete [] ptr;

ptr=newptr;

size=2\*size;

template<class T>

Node<T>\* Node<T>::return\_last\_remove(Node \*list){

Node \*temp=list;

while (temp->next->next!=NULL)

{

temp=temp->next;

}

printf("\n");

cout<<"Last with remove: "<<temp->next->data<<endl;

temp->next=NULL;

// temp->prev=NULL;

delete (temp->next);

return list;

}

Direct access table:distinc keys,oper. in O(1)

Hash functions:unsuccess. Search chaningO(i+a)

Avg,best O(1)

ADS