Data Structures & Algorithms

Rahil Prakash, 2026 Spring Semester

Contents

1. Data Structures and Algorithms Lecture 1

1.1. Sorting

Arrange (or order) a list of numbers in ascending or descending order.

1.2. Examples/Applications

- Sorting students by marks
- Sorting colleges/companies by their rankings
- Sorting videos by their relevance
- Sorting flights by their ticket price.
- Arranging students by their name or roll numbers
- Sorting patients by their weight or height or age

Input: A

0	1	2	3
27	36	2	7

Output: A

0	1	2	3
2	7	27	36

1.3. Aim

To devise a good algorithm to sort these numbers in ascending order

1.3.1. Brute Force

All possible arrangements There are `n!` possible orderings.

Note

Given an array of size `n` only `n-1` comparisons are needed to verify sorting `A[i-1] <= A[i]` ∀ integers `1 <= i <=n`

1.3.2. Bubble sort

```
swapped = 1;
while (swapped == 1) {
  swapped = 0;
  for(i = 1; i < n; i++) {
    if (A[i-1] > A [i]) {
      swap(A[i-1], A[i]);
      swapped = 1;
    }
  }
  // n-- for better efficiency
}
```

Bubbles an element to the top of the array. Guaranteed to sort it in `n-1` runs of the inner loop. Time complexity is `O(n²)` As the last n elements are guaranteed to be sorted after n runs of the for loop, we can introduce an `n--` term after the for loop as there is no more need to check it. Running time without the efficiency improvement is `(n-1)*n` and with it is `(n-1)n/2`

1.3.2.1. Puzzle

> A number in a series is considered a rabbit if it moves forward in position after being sorted and a tortoise if it moves back. Find a way to make an arrangement of 8 non-equal numbers such that there are 7 rabbits A: sort in ascending order and then send first element to the end.
>
> Find a way to make an arrangement such that there are 0 rabbits and non zero tortoises

A: This is impossible, because for the last element it can only be in its place to not be a rabbit, and from there by induction it is impossible to have a tortoise without having a rabbit.

Important

Does the number of rabbits and tortoises decrease after each iteration of the outer loop of the bubble sort algorithm?

A: Rabbits may increase (5 2 3 4 1) but tortoises will always stay the same or decrease because an element in a single iteration may jump ahead multiple places but can only jump back one step.

If the number of tortoises has increased, then an element in its correct position has jumped ahead, meaning that the element ahead of it is smaller than it, and all elements before it are smaller, which is impossible.

2. Insertion sort

27	26	2	7
2	7	26	27

It works like sorting a deck of cards

```
for (i = 1; i <=n; i++) {
  for (j = i; j > 0 && A[j-1] > A[j]; j--){
    temp = A[j];
    A[j] = A[j-1]
    A[j-1] = temp;
  }
}
```

In essence, after every outer loop, the first `i` numbers are sorted with respect to themselves

27 26 2 7 -> 26 27 2 7 -> 2 26 27 7 -> 2 7 26 27

Think of it as having unsorted cards in your right hand. You pick a card and insert it into the correct place in your left hand.

But the linear exchange swapping is inefficient. Then why do so?

This is because we >till need to get it there, so binary search won't help. You still need to have everything in the right place causing you to swap things, and binary search on a linked list is inefficient.

Data Structure	Search	Insert	Total
Array	O($\log_2$n)	O(n)	O(n)
Linked list	O(n)	O(1)	O(n)

2.1. Big O notation

It is used to represent the worst-case growth of time or space for an algorithm. It depends on the highest degree of the function defining the resource usage of the algorithm (time/space). For example an algorithm with running time `5n + 7` is `O(n)`

If you have a multi step algorithm its complexity is the complexity of the highest degree ex. Doing a linear insertion and then a binary search is `O(n) + O(log₂n) = O(n)`.

Important

If you are nesting algorithms complexity is *multiplicative*. If iterating through each element of a list of size n you do a binary search for it in a different list of size m, the time complexity is `O(n·log₂m)`

Linear search: O(n) Binary search: O($\log_2$n) (Technically it's log n but we use $\log_2$n for clarity; it divides search area in half per step)

3. Lecture 3

3.1. Comparing Sorting Algorithms

3.1.1. Bubble Sort

27	26	7	2	37	39	32	3	14
2	3	7	14	26	27	32	37	39

26 7 2 27 37 32 3 14 39 -> 7 2 26 27 32 3 14 37 39 Last 1 elements sorted -> Last 2 elements sorted These are sorted according to the final output

For the worst case `6 5 4 3 2 1 -> 1 2 3 4 5 6`

The number of swaps = number of pairs of elements = `nC2`

3.1.2. Insertion Sort

27	26	7	2	37	39	32	3	14
2	3	7	14	26	27	32	37	39

26 27 7 2 37 39 32 3 14 -> 7 26 27 2 37 39 3 14 First 2 elements sorted -> First 3 elements sorted These are sorted according to themselves.

For the worst case `6 5 4 3 2 1 -> 1 2 3 4 5 6`

The number of swaps = number of pairs of elements = `nC2`

The advantage of Insertion Sort is that it is an *online* algorithm, as in it can easily sort data coming in one by one

3.1.3. Selection Sort

27	26	7	2	37	39	32	3	14
2	3	7	14	26	27	32	37	39

Works by picking the minimum element and moving it to the left.

`2 27 26 7 37 39 32 3 14` -> `2 3 27 26 7 37 39 32 3 14`

```
for (int i = 0; i <= n-2; i++) {
  min = A[i], pos = i;
  for (int j = i + 1; j <= n - 1; j++) {
    if (A[j] < min) {
      min = A[j], pos = j;
    }
  }
 swap(A[i], A[pos]);
}
```

For the worst case, the number of swaps = `n-1`

We see that Bubble Sort has no advantages over the other algorithms.

3.2. Combining Two Sorted Lists

2	3	4	7	11	12	15	16
1	3	7	14	26	27	32	37

We use a 2 pointer approach. Each pointer initially points to the first element of each list. We compare both elements and add the appropriate element to the final list, and increment the pointer of that list.

We continue to do so until we get to the end of one of the lists. At that point the remaining portion of the other list can be appended to the end

A is of size m, B is of size n. `i: 0->m, j: 0->n` Running time = k(m+n) = O(m+n)

3.3. Merge Sort

We divide the list to the point that it is trivial to sort (1 or two elements) and then merge it together

`4 2 3 1 -> (4 2) (3 1) -> (2 4) -> (1 3) -> 1 2 3 4`

Number of comparisons: `∑ (2^k - 1) * n/2^k; 1 <= k; n >= 2^k`

```c
int* merge(int A[], int B[], int m, int n) {
  int C[m + n];
  int i = 0, j = 0;
  for (int k = 0; k < m + n; k++) {
    if (i > m - 1) {
      C[k] = B[j]; j++;
    } else if (j > n-1) {
      C[k] = A[i]; i++;
    } else if (A[i] < B[j]) {
      C[k] = A[i]; i++;
    } else {
      C[k] = B[j]; j++;
    }
  }
  return C;
}

void mergeSort(int X[], int n) {
  if (n <= 1) return;
  p = n/2;

  mergeSort(X, p);
  mergeSort(X + p, n - p);

  int* temp = merge(X, X + p, p, n - p);

  for (int i = 0; i < n; i++) X[i] = temp[i];

  free(temp);
}
```

This algorithm has a time complexity of `O(nlog₂n)`

4. Quick Sort

4.1. Idea

Quick Sort is a **divide-and-conquer** sorting algorithm.

Instead of dividing the array into equal halves (like Merge Sort), it:

1. Chooses a **pivot element**
2. Partitions the array such that:
   - Elements **less than or equal to the pivot** are on the left
   - Elements **greater than the pivot** are on the right
3. Recursively applies the same process on the left and right subarrays

After partitioning, the pivot is in its **final sorted position**.

4.2. Example

Input:

27	26	7	2	37	39	32	3	14
7	2	3	**14**	27	26	37	39	32

Choose pivot = **14**

Recursive calls:

- Left subarray: `7 2 3`
- Right subarray: `27 26 37 39 32`

Final Output:

2	3	7	14	26	27	32	37	39

4.3. Partitioning

Partitioning is the core operation of Quick Sort.

A pivot element is chosen and the array is rearranged so that all elements smaller than or equal to the pivot appear before it and all greater elements appear after it.

4.3.1. Lomuto Partition Scheme

Pivot is chosen as the **last element**.

```
int partition(int A[], int low, int high) {
  int pivot = A[high];
  int i = low - 1;

  for (int j = low; j < high; j++) {
    if (A[j] <= pivot) {
      i++;
      swap(A[i], A[j]);
    }
  }
  swap(A[i + 1], A[high]);
  return i + 1;
}
```

After partitioning:

- `A[p]` is in its **correct sorted position**
- Elements to the left of `p` are ≤ pivot
- Elements to the right of `p` are > pivot

4.4. Algorithm

```
void quickSort(int A[], int low, int high) {
  if (low < high) {
    int p = partition(A, low, high);
    quickSort(A, low, p - 1);
    quickSort(A, p + 1, high);
  }
}
```

4.5. Properties

- In-place sorting algorithm
- Not stable
- Recursive
- Based on divide-and-conquer paradigm

4.6. Time Complexity

Case	Time Complexity
Best case	$O(n\log_2 n)$
Average case	$O(n\log_2 n)$
Worst case	$O(n^2)$

4.6.1. Worst Case Explanation

Worst case occurs when the pivot chosen is always the **smallest** or **largest** element.

Example:

`1 2 3 4 5 6` This produces subproblems of sizes `n-1` and `0`.

Recurrence relation:

$$T(n) = T(n-1) + O(n)$$

Which results in:

$$T(n) = O(n^2)$$

4.7. Number of Comparisons

- Best and average case: $O(n\log_2 n)$
- Worst case: `nC2` comparisons

4.8. Improving Performance

4.8.1. Randomized Quick Sort

Randomly choosing the pivot reduces the probability of worst-case behavior.

```c
int randomPartition(int A[], int low, int high) {
  int r = low + rand() % (high - low + 1);
  swap(A[r], A[high]);
  return partition(A, low, high);
}
```

4.9. Comparison with Other Sorting Algorithms

Algorithm	Best	Average	Worst	Extra Space	Stable
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No
Merge Sort	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	Yes
Quick Sort	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)$	No

4.10. Key Observations

Important

Quick Sort is usually faster in practice than Merge Sort despite having the same average time complexity.

Reasons:

- Better cache locality
- No extra memory allocation
- Smaller constant factors

4.11. When to Use Quick Sort

- When in-place sorting is required
- When average-case performance is more important than worst-case guarantee
- Widely used in real-world libraries and systems

5. Lecture 5

5.1. Table of contents

5.2. Sort 4 elements using 4 comparisons

It is impossible to guarantee that you can sort 4 numbers using 4 comparisons.

Four elements: `a b c d` -> 4! = 24 possible combinations for final order The optimal comparison will split the possibilities into 2

24 possibilities -> 12 -> 6 -> 3 -> 1.5 (1 or 2)

So, in the worst case you will need 5 comparisons

2^k >= n! -> k >= log$_2$(n!) -> k >= (1/4) n log$_2$(n)

```
2^k >= n x (n-1) x (n-2)x ... (n/2 - 1) ... 3 x 2 x 1
-> 2^k >= (n/2)^(n/2) x 2^(n/2)
k >= (n/2)log(n/2) + (n/2)log 2 >= (n/4)(log(n/2) + log 2) >= (n/4)log(n/2)
```

Important

Every comparison-based sorting algorithm makes at least (1/4) n log$_2$(n) comparisons in the worst case

5.3. Stacks

FILO (First-In Last-Out) Data Structure

5.3.1. Basic Stack Operations

```c
int isEmpty(int top) {
  if (top == -1) {
    return 1;
  }
  return 0;
}

int isFull(int top, int max) {
  if (top == max-1) {
    return 1;
  }
  return 0;
}
```

- Push

```c
void push(stack[], top, max, item) {
  if (top == max - 1) {
    printf("Stack overflow");
    return;
  } else {
    stack[++top] = item;
  }
}
```

- Pop

```c
int pop(stack[], top) {
  if (isEmpty) {
    printf("Empty");
    return -1;
  }
  return stack[top--];
}
```

5.4. Checking For Balanced Parentheses

(a + b)(c + d) - (2a(3c −4b) + 6c) -> Good 1 12 2 3 4 4 3 (2c + 43)) -> Wrong

5.4.1. Rules For Balanced Parentheses

- Every open parenthesis is associated with exactly one closed parenthesis.
- At every step while scanning the expression from left to right, the number of open parentheses >= closed parentheses.

5.4.2. Algorithm

The algorithm for doing this with a stack is as follows:

For each open parenthesis we push onto the stack and pop it for each closed.

If you ever try to pop an empty stack or if the stack isn't empty at the end, the parentheses aren't balanced

6. Lecture 6

6.1. Table of Contents

6.2. Evaluating Mathematical Expressions

6.2.1. Evaluation

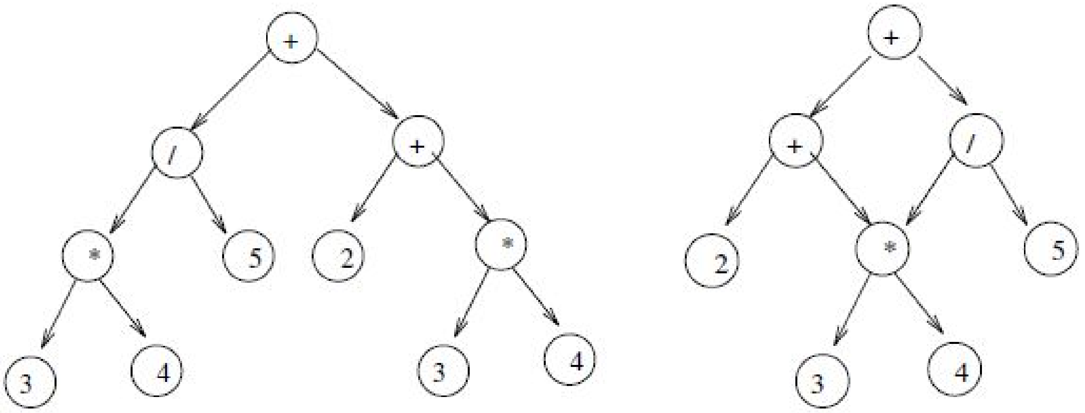


Figure 5.17: Expression $2 + 3 * 4 + (3 * 4)/5$ as a tree and a DAG

6.2.2. Infix Notation

```
A + ((B · C) - (D/E^F) · G ) · H
```

```
(a+b) left subtree · operator · right subtree
```

Readable by humans

6.2.3. Postfix Notation

Readable by computers/calculators. Used in languages like LISP

```
(ab+) left subtree · right subtree · operator
```

6.2.3.1. Advantages of Postfix

- No parenthesis
- No operator precedence
- No associativity

Infix	Postfix
A + ((B · C) - (D/E^F) · G) · H	ABC · DEF^/G · -H · +

Postfix notation was initially called RPN (Reverse Polish Notation) named after the Polish scientist Jan Lucasiewicz. It was first used in a calculator in 1938

6.2.3.2. Rules for Postfix Evaluation

1. Operand -> Postfix

2. Open Bracket -> Push to Stack
3. Closed Bracket -> Pop everything one by one up to and including an opening bracket and put in postfix
4. Operator -> Follow the chart, then push the current operator to stack

What to do?	Precedence	Associativity
Pop	tos > op	-
Pop	tos = op	left -> right
Push	tos = op	right -> left
Push	tos < op	-

Column1	Column2	Column3
A	(empty)	A
+	+	A
(	+(	A
B	+(	AB
·	+(·	AB
C	+(·	ABC
-	+(-	ABC ·
(	+(-(	ABC ·
D	+(-(	ABC · D
/	+(-(/	ABC · D
E	+(-(/	ABC · DE
^	+(-(/^	ABC · DE
F	+(-(/^	ABC · DEF
)	+(-	ABC · DEF ^/
·	+(-·	ABC · DEF ^/
G	+(-·	ABC · DEF ^/ G
)	+	ABC · DEF ^/ G ·-
H	+	ABC · DEF ^/ G ·- H

End of expression -> Pop stack

Final Expression: `ABC · DEF^/G · -H · +`

Another way of doing it is traversing the tree top down and adding an element only when you encounter it for the last time

6.2.3.3. Puzzle

Q: Convert to Postfix: `(56 + 12) · 3 - 4`

A: `56 12 + 3 · 4 -`

Q: `A^B^C`

Character	Stack	Postfix
A	(empty)	A
^	^	A
B	^	AB
^	^^	AB
C	^^	ABC

End of expression -> Pop stack

A: `ABC ^^`

6.2.3.4. Evaluating Postfix

Postfix Evaluation is done using a stack, scanning the expression left -> right

- Operand -> Push to Stack
- Operator:
  - Pop the top element (y)
  - Pop next element (x)
  - Calculate z = x operator y
  - Push z to stack

6.3. Queue

First-In-First-Out (FIFO) Data Structure

6.3.1. Queue Operations

- Insert (enqueue)
- Delete (dequeue)

7. Lecture 7

7.1. Queues

FIFO: First In First Out Data Structure

```
queue_insert(q[], front, rear,  max, item) {
  if (front == rear + 1 || front == 0 && rear == max - 1){
    print "Overflow"
  } else {
    if ( front == -1) front = 0, rear = 0;
    else if (rear == max-1) rear = 0;
    else rear = rear + 1;
    q[rear] = item;
  }
}
```

```
queue_delete(q[], front, rear, max) {
  if (front == -1) {
  print "Empty"
  }
  else {
    item = q[front];
    if (front == rear) {
      front = -1, rear = -1;
    } else if (front == max-1) {
      front = 0;
    } else {front++;}
    return empty;
  }
}
```

Effectively, `front` is the pointer to the earliest element inserted into the queue and `back` is the pointer to the latest element inserted. We do a wraparound when we get to the end and if the pointers cross, there is an overflow and we need to reallocate the queue.

8. Lecture 8

8.1. Doubly Ended Queues (Dequeues)

Insertion and deletion can happen at both ends.

Example : Hospital waiting list. In case of emergency removal is from rear instead of front.

-	6	12	-
-	6	12	5
-	-	12	5
-	-	12	-
-	7	12	-
-	7	12	9

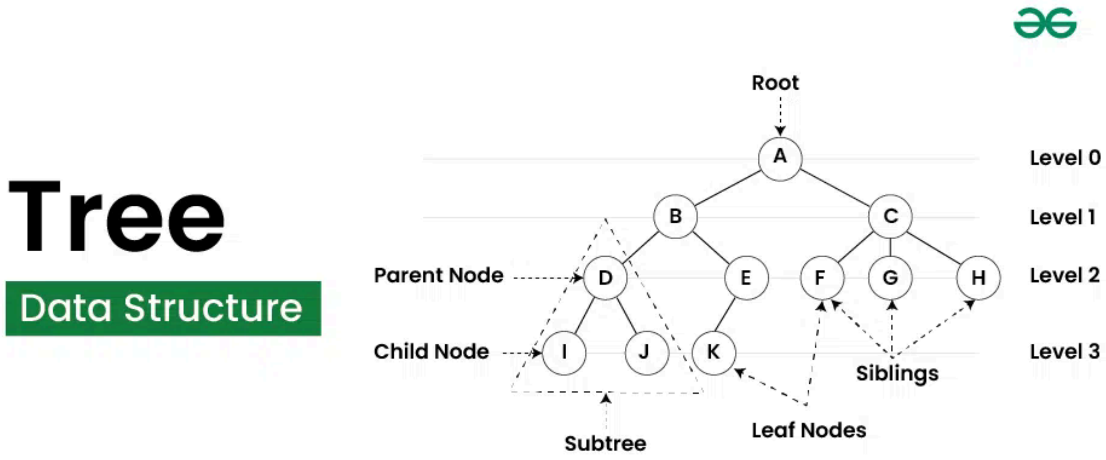
1. Add 5 to rear
2. Remove from front
3. Remove from rear
4. Add 7 to front
5. Add 9 to rear

```
insert_front(q[], front, rear, max, item) {
  if (front == rear + 1 || front == 0 && rear == max -1) {
    printf("Queue Full"\n);
  } else {
    if (front == -1) {
      front = 0; rear = 0;
    } else if (front == 0) {
      front = max-1;
    } else {
      front = front -1;
    }
    q[front] = item;
  }
}

delete_rear(q[], front, rear, max) {
  if (front == -1) {
    printf("Queue empty");
  } else {
    item = q[rear];
    if (front == rear) {
      front = rear = -1;
    } else if (rear == 0){
      rear = max - 1;
    } else {
      rear = rear - 1;
    }
    return item;
  }
}
```

8.2. Trees

Non linear data structure



8 Family Members

1-8 are nodes.

- 1 is the parent of nodes 2 and 3 and they are its children.
- 2 and 3 are siblings as they have the same parent.
- Root node is the only node without parents, in this case 1.
- Leaf nodes are the nodes without children.
- A node is an ancestor of another node if it lies on the path from it to the root node
- Node X is a descendant of node Y if Y is an ancestor of X.
- Depth: Distance from root node
- Height: Maximum depth amongst all nodes

8.2.1. Binary Tree

A tree in which each node has at most 2 children

8.2.2. Full Binary Tree

A tree in which each node has 0 or 2 children

8.2.3. Complete Binary Tree

All levels are filled in from top to bottom and left to right.

Important

It is not necessary for a complete binary tree to be full. e.g 1 - (2 - (4) 3) is not a full binary tree but it is a complete binary tree

8.2.4. Perfect Binary Tree

A full binary tree in which all leaves have the same depth.

> This means that it is also a complete binary tree

8.2.5. Balanced Binary Tree

Height of left and right subtrees of every node differ by at most 1.

In a complete binary tree the depths of all leaf nodes differ by at most 1.

However, Balanced Binary Trees are not necessarily Complete.

8.2.6. Degenerate / Pathological Binary Tree

Each node has at most one child

- Edge: Connection between two nodes. Number of edges = Number of nodes - 1
- For a full binary tree, `2h+1 <= n <= 2 ^ (h+1) - 1`
- Number of leaf nodes in a perfect binary tree is (n+1/2)
- Number of leaf nodes = i_2 + 1, i_2 = internal nodes of degree 2

9. Lecture 9

9.1. Implementation of Binary Trees

```
typedef struct node {
  int data;
  struct node* left;
  struct node* right;
} node;

node* root;
root = create(x);

create (x) {

  node* new;
  new->data = x;
  if (left subtree exists) {
    take value y for left node from user.
    new -> left = create(y);
    else new->left = null;
  }

  if (right subtree exists) {
    take value z for right node from user.
    new -> right = create(z);
    else new->right = null;
  }
}
```

9.2. How can we store a general tree (T) as a binary tree (B)?

The left-child right-sibling binary tree.

- Root of B is Root of T
- B's left child is T's first child, and B's right child represents T's next sibling.

9.3. Puzzle

p = number of null pointers

$p = i_1 + 2l = i_1 + 2l = n+1$

Important

$l = i_2 + i_1$

10. Lecture 11

10.1. Binary Search Trees

Binary Trees such that all elements are unique and all to the left are lesser and all to the right are greater than the current element.

10.2. Finding min and max in a BST

For min keep going left until you reach null pointer, that element is min. This is because all elements to the left are less than the element.

For max, keep going to the right.

10.3. Full Node

A node with exactly 2 children.

10.4. Non-full Node

A node with 0 or 1 children.

Important

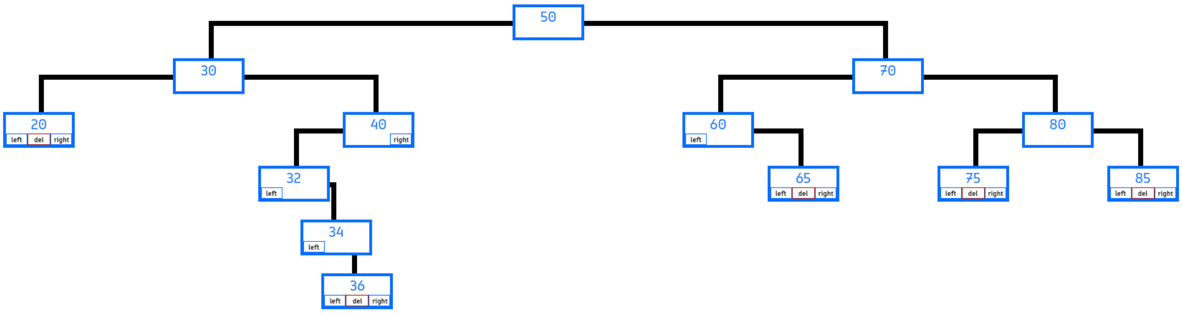
Min and Max are always Non-full nodes. This is because if they are full, they have a lesser and greater element, and thus are not the min or max.

10.5. In-order output

All left elements are printed, then node, then right. -> elements are printed in ascending order

Important

If y is the successor of x then x is the predecessor of y. The pair (x,y) is called a (predecessor, successor) pair.



Important

Both Nodes of a (predecessor, successor) pair can never be full nodes.

PROOF:

(i, j) -> (predecessor, successor)

1. If j comes before i, i will be in the left subtree of j.
2. If i comes before j, j will be in the right subtree of i
3. For every (pred, succ) pair, either pred is ancestor of succ or succ is ancestor of pred.

For 1: Since i is the max value < j it means that i is the max in the left subtree of j. Thus, i is NOT a full node, as at every subtree BST properties apply.

For 2: Since j is the min value > i it means that j is min in the right subtree of i. Thus, j is NOT a full node.

```
BST_SUCCESSOR(node) {
  if (node->right != NUL)
    return BST_MIN (node->right);
  // Keeps going until the subtree is left of parent
  p = node->parent;
  while (p != NULL and node == p->right) {
    node = p; p = node->parent;
    return p;
  }
}
```

```
BST_PREDECESSOR(node) {
  if (node->left != NUL)
    return BST_MAX (node->left);
  // Keeps going until the subtree is right of parent
  p = node->parent;
  while (p != NULL and node == p->left)
    node = p; p = node->parent;
  return p;
}
```

```
// Inserts at leaf
BST_INSERT(root, item) {
  y = NULL, x = root;
  while (x != NULL) {
    y = x;
    if (item < x->data)
      x = x->left;
    else
      x = x->right;
  }
  node z;
  z->parent = y;
  z->data = item;
  z->left = NULL;
  z->right = NULL;

  if (root == NULL) return z;
  if (item < y-> data)
    y->left = z;
  else
    y->right = z;
  return root;
}
```

10.6. Delete

Cases:

1. (trivial) 0 children
2. (easy) 1 child

3. (tricky) 2 children

```
BST_DELETE(root, item) {
  while (root -> data != item && root != NULL){
    if (root->data < item)
      root = root->right;
    else if (root->data > item)
      root = root->left;
  }
  if (root->data == item) {
    if (root->left!=NULL && root->right!=NULL){

    }
    else if (root->left!=NULL && root->right!=NULL) {

    }
    else {
      node p = root->parent;
      if (p->right == root)
        p->right == NULL;
      else
        p->left == NULL;
    }
  }
}
```

```
BST_DELETE(root, node) {
  if (node->left == NULL)
    SHIFT_UP(root, node, node->right);
  else if (node->right == NULL)
    SHIFT_UP(root, node, node->right);
  else {
    y = BST_SUCCESSOR(node);
    swap(y->data, node->data);
    BST_DELETE(root, y);
  }

}

// Only valid for Non-full nodes
SHIFT_UP(root, node, child) {
  if (node->parent == NULL)
    root = child;
  else if (node = node->parent->left)
      node->parent->left = child;
  else if (node = node->parent->right)
      node->parent->right = child;
}
```

11. Lecture 12

```
DELETE_NONFULL(root, node, child) {
  if (node->parent == NULL)
    root = child;
  else
    if (node == node->parent->left)
      node->parent->left = child;
    else
      node->parent->right = child;
  if (child != NULL) {
    child->parent = node->parent;
  }
  free(parent);
}
```

```
DELETE_FULL(root, node) {
  y = BST_SUCCESSOR(node);
  swap(node->data,y->data);
  if (y->right == NULL)
    child = y->right;
  else child = y->right;
  DELETE_NONFULL(root, y, child);
}
```

12. Lecture 13

- Finding min/max -> O(n) unsorted, O(1) sorted
- Maintaining min/max
  - Inserts are happening -> O(1) per element
  - Deletes are happening -> O(n) per element

12.1. Heaps

- Unlike BSTs, Heaps are binary trees that have both children > node (min-heap).
- This means all ancestors < their descendants. Doesn't apply to other descendants.
- Complete (or left justified) binary tree.

12.1.1. Left-justified (or Complete) binary tree

All nodes are inserted from top to bottom, and left to right.

For height h, 2^h <= no. of nodes <= 2^(h+1) - 1

-> h <= $\log_2(n)$

n	h
1	0
2	1
4	2
8	3
16	4

The minimum element of a min-heap is always the root node.

12.1.2. Insertion into Heap

We insert into the appropriate position at bottom, such that it maintains left-justified property.

Then, if inserted < parent, swap them. Repeat until node < parent or parent = NULL

Number of swaps performed <= h <= $\log_2(n)$

Insertion time <= $O(\log_2(n))$

12.1.3. Deletion from Heap

Take element and swap it with latest element. Delete that element, and fix the tree.

If it is the problem with children, swap it with the lesser one.

Number of swaps performed <= h <= $\log_2(n)$

Deletion time <= $O(\log_2(n))$

Data Structure	Insertion	Deletion
Array	O(1)	O(n)
Heap	$O(\log_2(n))$	$O(\log_2(n))$

This is better than just a sorted array as for each operation you need O(n) swaps.

12.1.4. Heap as an Array

Let array be A. Root is `A[0]`

For any node with index i, its children are `A[2 * i + 1]` and `A[2 * i + 2]`

Similarly, for a node, its parent is `(i - (2 - (i % 2))/2` or `(i-1)/2)`

Important

A sorted array (ascending order) is a valid representation of a min-heap.

Thus, as construction from a random array is $O(n\log_2(n))$, sorting the array is a viable way to start.

13. Lecture 14

13.1. Notation

n: number of nodes N: Next power of two

13.2. Operations on a heap

- insert -> $O(\log_2 n)$
- delete-min and delete-max -> $O(\log_2 n)$
- heapify -> O(n)
- heap sort -> $O(n\log_2 n)$

13.3. Heapify

Given an array, convert it to a min-heap

13.3.1. Sorting

Simply sort the array. $O(n\log_2 n)$

13.3.2. Heapify top-down

- Start from the top and go downwards
- Ensure everything above current node is a valid heap.
- Once you reach bottom, everything is valid.

Time Complexity

Since heap is a complete binary tree, depth of a node is at most $\lceil \log_2 n \rceil$. Each Node moves upwards at most $\log_2 n$ times, and there are n nodes Complexity is thus <= $n\log_2 n$ -> $O(n\log_2 n)$

Each left node undergoes $\log_2 n$ exchanges.

Total time = ½ N $\log_2 n$ >= ½ n $\log_2 n$

½ N $\log_2 n$ >= ½ n $\log_2 n$

13.3.3. Heapify bottom-up

- Start from bottom and go upwards.
- Ensure that everything below current node satisfies heap property.
- Once you reach top heap is created.

Time Complexity

Since heap is a complete binary tree, depth of a node is at most $\lceil \log_2 n \rceil$. Each Node moves downwards at most $\log_2 n$ times, and there are n nodes Complexity is thus <= $n\log_2 n$

Each parent of leaf (P(leaf)) undergoes at most one exchange in a step

P(P(leaf)) undergoes at most two exchanges in a step, and so forth

Leaf nodes = N/2 (0 exchanges)

P(Leaf) = N/4 (1 exchanges)

P(P(Leaf)) = N/8 (2 exchanges)

P(P(P(Leaf))) = N/16 (3 exchanges)

```
X = N/2 · 0 + N/4 · 1 + N/8 · 2 + N/16 · 3 + ...
```

```
2X = N/2 · 1 + N/4 · 2 + N/8 · 3 + N/16 · 4 + ...
```

```
2X-X = X = N/2 + N/4 + N/8 + N/16 + N/32 ...
```

```
X <= N
```

Thus complexity is O(N)

13.4. Heap Sort

Keep performing delete-max until there is nothing left to delete.

We delete max element and insert it into end of sorted array.

In an actual heap, we swap it to the end of the array, reduce the size, and heapify.

This heapify will take $\log_2 n$ steps at worst. Thus total time complexity is

$\log_2(n) + \log_2(n-1) + \log_2(n-2) + \log_2(n-3) \ldots + \log_2(2) = \log(n!) <= O(n\log_2 n)$

Heap sort <= (n + 1) + c(n $\log_2$n) ↓ ↓ Heapify Extraction

14. Lecture 15

14.1. Hashing

14.1.1. Heap Sort

- More swaps in general.
- Time complexity: O(n$\log_2$n)

14.1.2. Hashify

- Super fast data retrieval
- Hash table = Array (usually)
  - Unsorted = O(n) : Slow
  - Sorted = O($\log_2$n) : Fast
  - Hash Table = O(1) : Superfast
- Retrieval using arrays is O(1)
- Arrays are Fixed/Rigid/Static data structure

14.2. Uses of Hashing

- World Wide Web [Google Search]
- Train/Flight booking systems, stock market websites [IP address]
- Search a word in a pdf file.
- Playing a live real-time multi-player game, possibly with high-end graphics
- Cryptographic password storage.
- Fingerprints
- Database indexing
- Caching systems

14.3. Methods of Hashing

- EVE = 5 + 22 + 5 = 32 = 3 + 2 = 5
- FEZ = 6 + 5 + 26 = 37 = 10 = 1

Create a function $f(x)$: Takes in a string and returns a number.

This function is called a *Hash function.*

Collisions

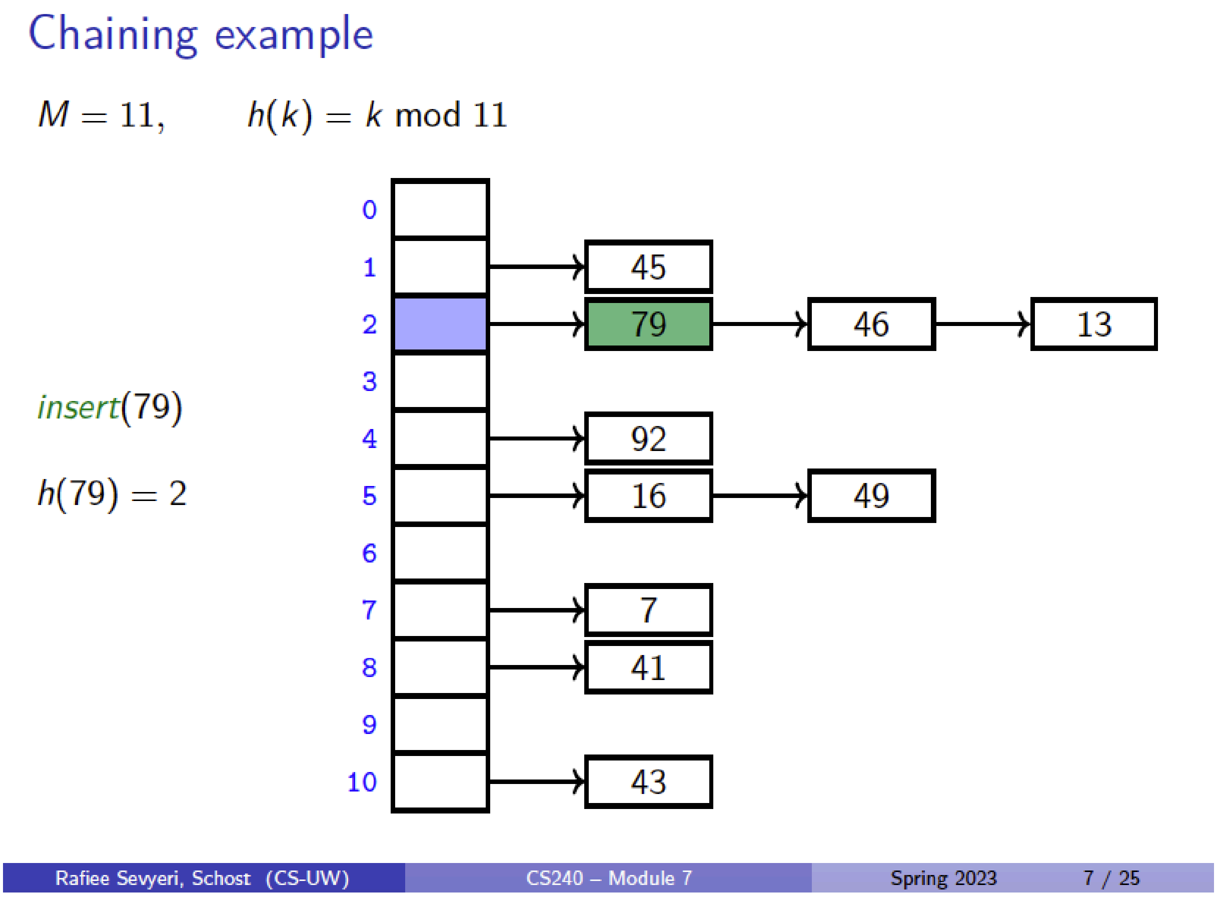
Situations where $X_1 \neq X_2$ but hashes are equivalent.

Load Factor

$(\alpha)\ \text{Load Factor} = \frac{\text{Number of Items Stored}}{\text{Size of Array}}$

Open addressing ($\alpha < 1$)	Closed addressing ($\alpha > 1$)
Items can go to any location	Items can go only to their prescribed location
Elements are stored within the array itself	Elements are stored within external data structures
Collision handling is done by linear probing, quadratic probing	Collision handling is done via linked lists and chaining
Quicker retrieval	Better at handling deletions (O(1))

14.4. Closed Addressing (Chaining)



Chaining example

$M = 11, \qquad h(k) = k \bmod 11$

insert(79)

$h(79) = 2$

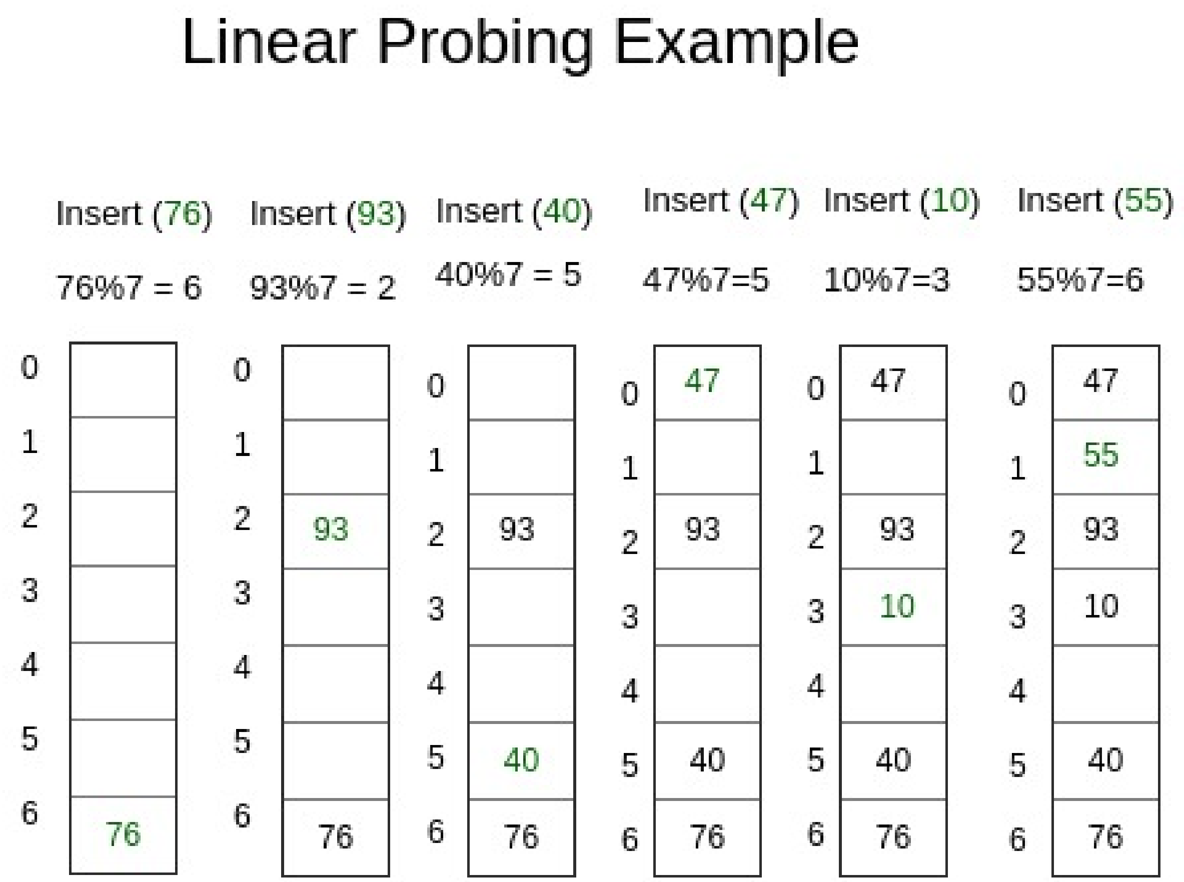
Rafiee Sevyeri, Schost (CS-UW) CS240 – Module 7 Spring 2023 7 / 25

In the worst-case scenario we have α elements in each chain. If α gets too big we remake the hashmap and choose a new hashing function

14.5. Open Addressing

14.5.1. Linear Probing

If collision occurs we move on to the next empty element (wraparound)



Primary Clustering

Items end up bunching together in linear probing

These clusters also tend to compound quite quickly.

This is bad because you now have increased search times for lookups and inserts, degrading complexity from O(1) to O(n).

14.5.2. Quadratic Probing

If a collision occurs, the next position is looked for as follows:

$h(k, i) = ai^2 + bi + h(k)$ where $h(k)$ is the auxiliary hash function.

14.5.3. Forms of Open Addressing

- Linear Probing
- Plus 3 Rehash
- Quadratic Probing
- Double Hashing: In case of collision a second hash function is applied to the first

For example, $H(k, i) = h_1(k) + ih_2(k)$

- If $\alpha = 1$, it is theoretically possible to design a perfect hash function (0 collisions)

Important

What is a good hash function?

- Minimizes collisions
- Uniformly distributes the hash values
- Easy to compute

- Resolve collisions efficiently

14.6. Universal Hashing

- Universal family of hashing functions
- Choose one hash function uniformly at random
- Family: $H, \text{ choose } h \epsilon H$

$$h_{a,b}(x) = [(ax + b) \mod p] \mod |A|$$

Note

When $x_1 \neq x_2$, then the number of different hash function $h_{a,b}(x_1) = h_{a,b}(x_2)$ is at most $\frac{|H|}{|A|} \Longrightarrow$ If A is large, the probability of a collision is very small

15. Lecture 16

15.1. Why Binary Search Trees?

> What is the height of a BST with n nodes?

Height varies from $\lfloor \log_2 n \rfloor$ to $n - 1$

Time complexity of Insert and Delete is O(H) so the lower bound of H is the best case scenario.

best case	⌋log$_2$n⌊
avg case	$\frac{n}{2}$
worst case	$n - 1$

15.2. Proof of log$_2$n on avg

For n elements there exist `n!` ways of constructing said binary tree All elements are coming in random order

Note

The Inorder traversal of all n! BSTs is the same (Ascending order)

Assumption: A corner element does not occur 3 consecutive times, or every 3 consecutive elements include at least one middle element.

The middle 98% is middle elements, and the left and right 1% are the corner elements.

Let us assume the first element is a middle element, with L subtree bigger than R subtree.

Let us assume the next two elements are corner elements, i.e don't have two subtrees.

After 3 steps, the remaining nodes are at most $\frac{99}{100}n$

After 3k steps, the remaining nodes are at most $\left(\frac{99}{100^k}\right)n$

$k = \log_n\left(\frac{100}{99}\right)$

15.2.1. Issues with this proof

- Assumption does not cover all cases
- Abuse of the Big O notation. In most practical cases the constant is bigger than the function itself
- Grains of rice proof

$$X_{i,j} = \begin{cases} 1 \text{ if node } i \text{ is an ancestor of node } j \\ 0 \text{ otherwise.} \end{cases}$$

In how many of the 10! BSTs of 1-10 is node 7 an ancestor of node 6?

Relationship of all other nodes with 6 and 7 is identical.

Thus, for all permutations where 6 appears before 7, 6 is an ancestor of 7 ($X_{6,7} = 1$) and vice versa.

Thus, $Pr\left[X_{7,6} = 1\right]$ = $\frac{1}{2}$ because for all permutations where 6 is before 7, there exists a permutation where the 6 and 7 are swapped and 7 comes before 6

$Pr\left[X_{8,6} = 1\right]$?

Out of 3! permutations `867` and `876` are valid orders for ancestry

So, $Pr\left[X_{8,6} = 1\right] = \frac{1}{2}$

$Pr\left[X_{9,6} = 1\right]$?

If from 678 any occur before, 9 and 6 are split apart into different subtrees from that element. Thus probability is 1/4

What is the expected depth of node 6?

$\frac{2+3+4+2+4+\dots}{10!}$

D_j : Depth of node j

Expected depth = Number of ancestors

$D_6 = X_{1,6} + X_{2,6} + \dots + X_{10,6}$

$$ \begin{aligned} \mathbb{E}[D_{6}] &= \mathbb{E}[X_{1,6} + X_{2,6} + \cdots + X_{10,6}] \ &= \mathbb{E}[X_{1,6}] + \mathbb{E}[X_{2,6}] + \cdots + \mathbb{E}[X_{10,6}] \ &= \Pr[X_{1,6} = 1] + \Pr[X_{2,6} = 1] + \cdots + \Pr[X_{10,6} = 1] \ &= \left(\frac{1}{6} + \frac{1}{5} + \frac{1}{4} + \frac{1}{3} + \frac{1}{2}\right)

- \left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5}\right) \end{aligned} $$

Harmonic Sum (H_n) = $log_e n + \gamma$ = $log_e n + 0.5772$

$$\begin{aligned} \text{Average depth} &\le 2\left(\frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n/2}\right) \\ &\le 2\left(\ln\frac{n}{2}\right) + 1.15 \\ &\le 2.88(\log_2 n - \log_2 2) \\ &\le 2.88 \log_2 n \end{aligned}$$