CMSC420 Advanced Data Structures

Michael Li

May 22, 2020

Contents

1 Lists

```
init() => initializes list
get(i) => returns element at index i
set(i, x) => sets ith element to x
length() => returns number of elements in the list
insert(i, x) => insert x prior to element a_{i} (shifts indices after)
delete(i) => deletes ith element (shift indices after)
```

Sequential Allocation (Array): when array is full, increase its size but a constant factor (e.g. 2). Amortized array operations still O(1)

Linked Allocation (Linked List)

Stack(push, pop): on on end of the list
Queue(enqueue, dequeue): insert at tail (end) and remove from head (start)
Deque(combo stack and queue): can isnert and remove from either ends of list
Multilist: multiple lists combined 1 aggregate structure (e.g. ArrayList)

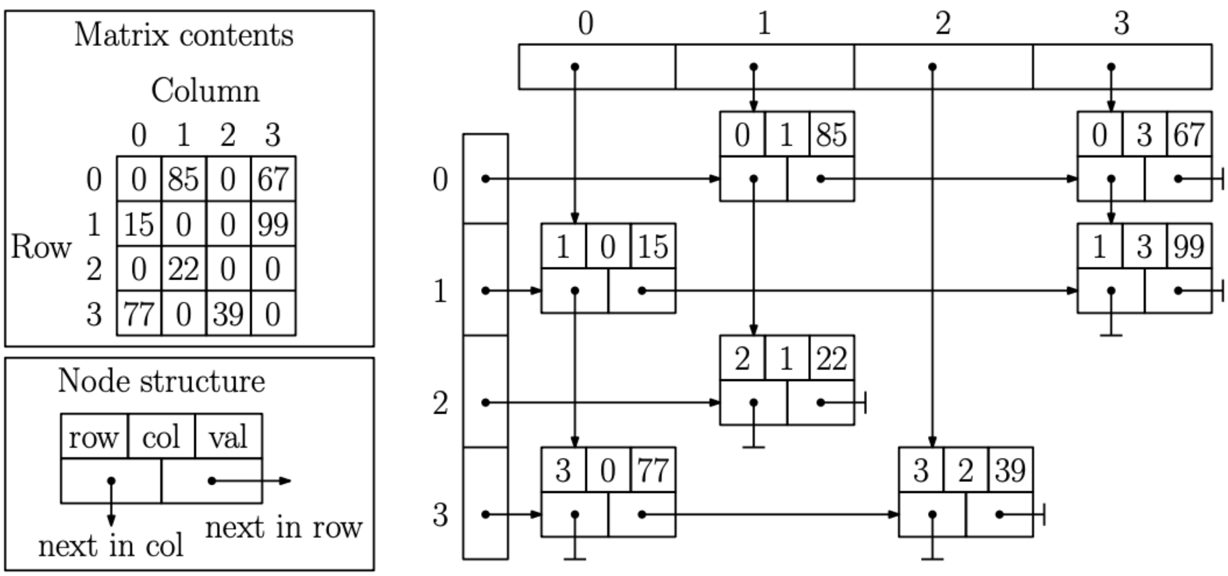

Fig. 2: Sparse matrix representation using a multilist structure.

Sparse Matrix: create 2n linked lists for each row and col
Each entry stores a row index, col index, value, next row ptr, and next col ptr

2 Trees

Free Tree: connected, undirected graph with no cycles (like MST)
Root Tree: each non-leaf node has ≥ 1 children and a single parent (except root)
Aborescence = out-tree Anti-arborescence = in-tree
Depth = max # of edges of path from root to a node

One way to represent tree is to have a pointer to first child and then a pointer to next sibling

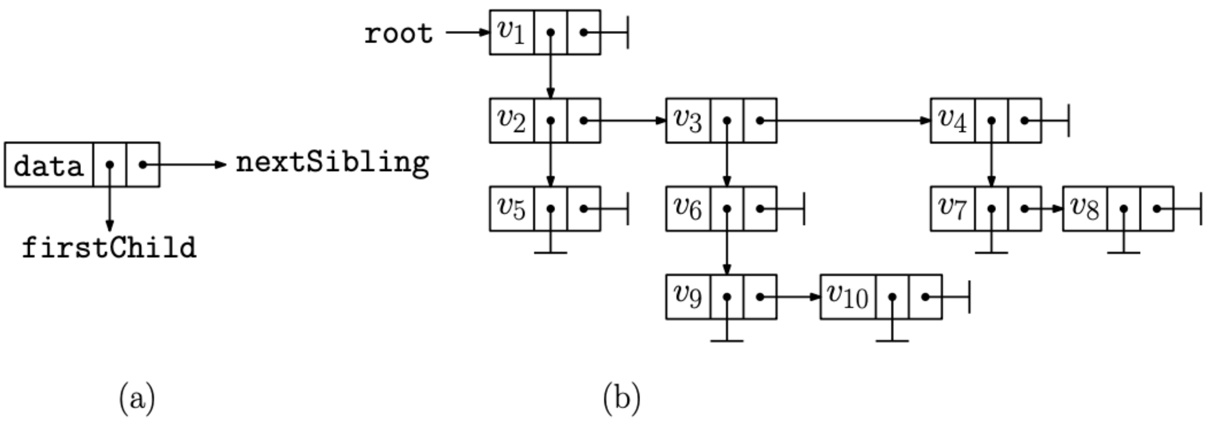

Fig. 3: Standard (binary) representation of rooted trees.

Binary Tree: rooted, ordered tree where each non-leaf node has 2 possible children (left, right)
Full Tree: All nodes either have 0 children or 2 children
Can make full binary tree by extending tree by adding external nodes to replace all empty subtrees

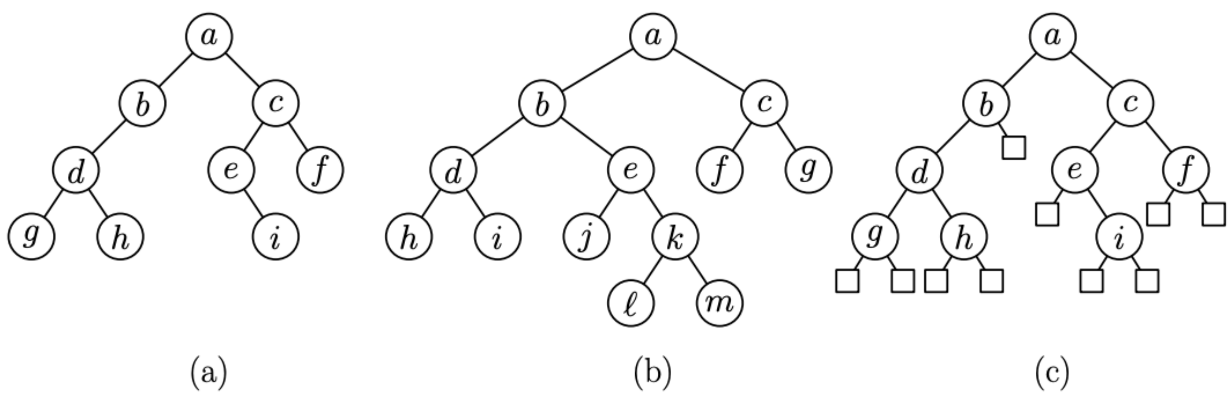

Fig. 4: Binary trees: (a) standard definition, (b) full binary tree, (c) extended binary tree.

```
class BinaryTreeNode<E> {
  private E entry;
  private BinaryTreeNode<E> left;
  private BinaryTreeNode<E> right;
  ...
}
```

In-order traversal: left, root, right
Pre-order traversal: root, left, right
Post-order traversal: left, right, root

If there are n internal nodes in an extended tree, there are n+1 external nodes

Proof by induction: Extended tree binary tree with n internal nodes has n+1 external nodes has 2n+1 total nodes

Let x(n) = number of external nodes given n internal nodes and prove x(n) = n + 1

Base Case x(0) = 1 a tree with no internal nodes has 1 external node

IH: Assume x(i) = i + 1 for all i $\leq$ n - 1

IS: let n_L and n_R be the number of nodes in Left and Right subtrees

x(n) = $(n_L + 1) + (n_R + 1) = (1 + n_L + n_R) + 1$ = n + 1 external nodes

so n + 1 (external) + n (internal) = 2n + 1

Moreover, about 1/2 of nodes of extended Binary Tree are leaf nodes

Threaded Binary Tree: Give null pointers information about where to traverse next

If left-child = null then stores reference to node's inorder predecessor

If right-child = null then stores references to node's inorder successor

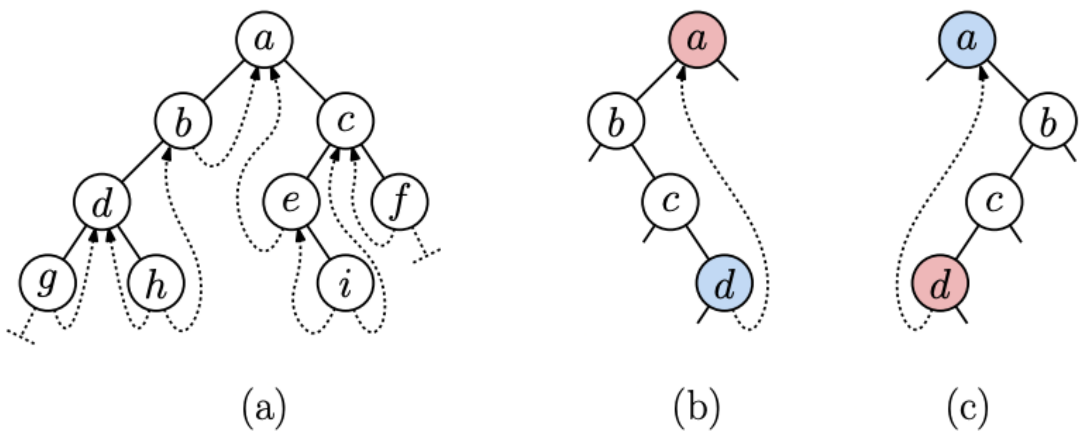

Fig. 6: A Threaded Tree.

```
BinaryTreeNode inOrderSuccessor(BinaryTreeNode v) {
BinaryTreeNode u = v.right;
if(v.right.isThread) return u;
while(!u.left.isThread) u = u.left;
return u;
}
```

if v's right-child is a thread, then we follow thread.

Otherwise go through v's right child and iterate through left-child links

Complete Binary Tree: represented using sequential allocation (array) because no space is wasted

number of nodes is inbetween 2^h and $2^{h+1} - 1$

```
leftChild(i): if(2i <= n) then 2i else null;
rightChild(i): if (2i + 1 <= n) then 2i + 1 else null;
parent(i): if (i >= 2) then [i/2] else null;
```

3 Dictionaries

```
void insert(Key x, Value v) => if key exists, exception is thrown
void delete(Key x) => if key does not exist, exception thrown
Value find(Key x) => return value associated with key or null if not found
```

Array representation:
Unsorted array has O(n) search and delete, O(1) insert although we need O(n) to check for duplicates
Sorted Array has O(logn) search and O(n) insertion and deletion
Binary Search Tree Representation (left < root < right):

```
//Recursive
Value find(Key x, BinaryNode p) {
  if (p == null) return null;
  else if (x < p.key) return find(x, p.left);
  else if (x > p.key) return find(x, p.right);
  else return p.val;
}

//Iterative
Value find(Key x) {
  BinaryNode p = root;
  while(p != null) {
    if (x < p.key) p = p.left;
    else if (x > p.key) p = p.right;
    else return p.value;
  }
  return null;
}
```

O(n) search for degenerate tree, O(logn) search for balanced tree
Can use extended BST to give info that target key is inbetween inorder predecessor and inorder successor
Insert: search for key and if found throw exception else we hit a null and insert there

```
BinaryNode insert(Key x, Value v, BinaryNode p) {
  if (p == null) p = new BinaryNode(x, v, null, null);
  else if (x < p.key) p.left = insert(x, v, p.left);
  else if (x > p.key) p.right = insert(x, v, p.right);
  else throw DuplicateKeyException;
  return p;
}
```

Either tree is empty so return new node or we return the root of the original tree with the added node
O(n) insert for degenerate tree, O(logn) insert for balanced tree
find a replace with inorder successor (aka leftmost on right subtree)

```
BinaryNode delete(Key x, BinaryNode p) {
  if (p == null) throw KeyNotFoundException;
  else
    if (x < p.data)
      x.left = delete(x, p.left);
    else if (x > p.data)
      x.right = delete(x, p.right)
    else if (p.left == null || p.right == null)
      if (p.left == null) return p.right;
      else return p.left;
    else
      r = findReplacement(p);
```

```
      //copy r's contents to p
      p.right = delete(r.key, p.right);
}

BinaryNode findReplacement(BinaryNode p) {
  BinaryNode r = p.right;
  while(r.left != null) r = r.left;
  return ;
}
```

O(n) deletion for degenerate tree, O(logn) deletion for balanced tree