Стек и опашка

(++put) % = capacity // =!= get за опашка

1 Стек - LIFO - stack

- push, pop, peek (top), isEmpty
- може с list, vector, deque
- по дефолт ако не програмист използва deque
- stack<int> st; - дефолтно
  stack<int, list<int>> st; - ако искаме да подадем конкретен контейнер сами
- използване в: undo/redo, история на браузъри, съобщения в месинджър итн, рекурсия стека, компилатори

2 Опашка - FIFO - queue

- enqueue, dequeue, front, isEmpty

в stl са: push, pop, front, isEmpty

- може с list, deque -> без vector
- дефолтно е с deque
- използва се за: принтиране на файлове, пакети от данни по мрежата, schedules, игри, кафка

// арийска кафка - за разпределение на входящи данни, schedules, събития, task-ове

//list - O(1), vector - ~O(1)

// увеличаваме x2, пазим като сме на 1/4

3 Deque - Дек

от Quiz

1 Сложността на оператор [] за deque - O(1)

2 Обработка чрез с рекурсия - stack

3 Сложност с колко добавяне N елемента - O(N)

Дървета

- Йерархична структура от данни
- Свързан ацикличен граф
- Бързо добавяне и търсене
- ребро - edge or line or link

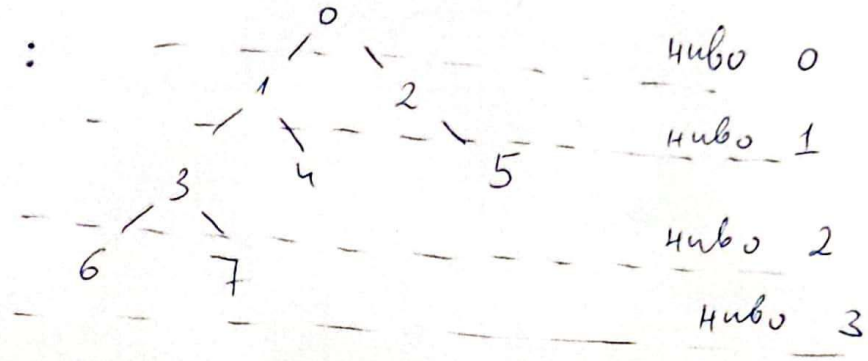
поддърво - subtree

корен - root

child - дете

лист - leaf - няма наследници

- ниво:



- абстрактна структура от данни
- двоично наредено дърво (двоично дърво за търсене) - Binary Search Tree
- двоично дърво - всеки връх има макс 2 наследника
- двоично наредено дърво - ляво по-малки, дясно по-големи
- $E = N - 1$ (e - ребра, n - върха) $\Rightarrow N = E + 1$
- височина → h и разклоненост → m (макс бр деца, които някой връх има) и в дървото има най-много m^h листа

// $2E = \sum_{v \in V} deg(v)$, deg - степен (колко ребра има до него)

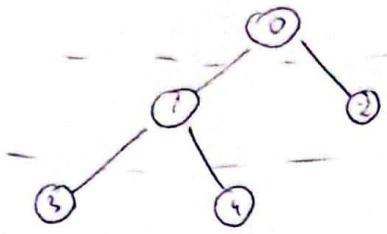
~~върха → листа~~

~~брой листа L = (de~~

~~Дефиниции~~ • НДТ - редица възли, всеки следващ е сред децата на предходния

• Широчина (разклоненост) - макс бр деца

•



depth 0, height 2
depth 1, height 1
depth 2, height 0

• In vector<int> tree[N]; ~~queue<int> q~~

- DFS (int v) {

```
cout << v << " ";
~~for auto x : tree~~
for(int i=0; i < tree[v].size(); i++)
    dfs(tree[v][i]);
```

}

- BFS (int v) {

~~queue<int> q; q.push(v);~~
~~while(!q.empty())~~

calculate height
for(i=0; i<height)
print(root, i);

```
Print(Node* root, int i){
  if(root == nullptr) return;
  if(i == 0) cout << root->data;
  else if(i > 1){
      print(root->left, i-1);
      print(root->right, i-1);
  }
}
```

Дървета

DFS Traversals - видове

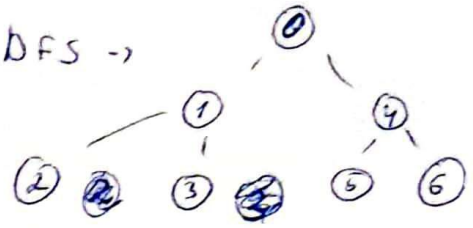
- inorder - ЛКД, ДКЛ // инфиксни
- preorder - КЛД, КДЛ // префиксни
- postorder - ЛДК, ДЛК // постфиксни

} зависи къде е корена

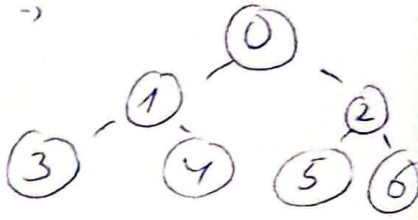
Search in BST

```
searchBST(Node* root, int v){
while(root){
        if(root->data == v)
                return root;
else if(root->data < v)
                root = root->r;
else if(root->data > v)
                root = root->e;
}
return root;
```

DFS ->



BFS ->



// може да е разбърканото ниво на BFS

Добавяне в BST

1 ~~Не~~ добавяне
2 пазим counter за повтаряне
3 добавяне вляво и дясно

Binary tree Properties

1 max бр nodes на ниво i = 2^i
2 max бр nodes на bt с височина h = $2^h - 1$
3 Възходящ ред на BT - inorder
4 Най-малката височина(ниво) при N nodes = $\log_2(N+1)$
(дълбочина)

Complexity

BST operation	Average case	Worst case
insert(key)	$\mathcal{O}(logn)$	$\mathcal{O}(n)$
remove(key)	$\mathcal{O}(logn)$	$\mathcal{O}(n)$
find(key)	$\mathcal{O}(logn)$	$\mathcal{O}(n)$
findMin()	$\mathcal{O}(logn)$	$\mathcal{O}(n)$
findMax()	$\mathcal{O}(logn)$	$\mathcal{O}(n)$

Сложност

implementation	guarantee		average case	
	search	insert	search hit	insert
sequential search (unordered list)	n	n	n	n
binary search (ordered array)	$\log n$	n	$\log n$	n
BST	n	n	$\log n$	$\log n$