Data Strcuture C++ Cheat Sheet

by eyan via cheatography.com/197714/cs/41730/

Linked List

Singly Linked List - Each node has data and an address field that contains a reference to the next node.
Doubly Linked List - There are two pointer storage blocks in the doubly linked list. The first pointer block in each node stores the address of the previous node.Then there is the data, and last you have the next pointer, which points to the next node. Thus, you can go in both directions (backward and forward).
Circular Linked List - The circular linked list is extremely similar to the singly linked list. The only difference is that the last node is connected with the first node, forming a circular loop in the circular linked list.

Operation: Traversing, Insertion, Deletion, Searching

Declaration

```
ListNode *dummyHead = new ListNo de(0);
ListNode *head;
```

Two Pointers

```
 ListNode *slow = head;
 ListNode *fast = head->next;
 while (slow != fast) {
      if (!fast || !fast- >next) return false;
      slow = slow->next;
      fast = fast->next->next;
}
```

Stack

`empty()` - O(1)
`size()` - O(1)
`top()` - O(1)
`push(e lement)` - O(1)
`pop()` - O(1)
`stackn ame 1.s wap (st ack name2)` - swap the contents of one stack with another stack O(1)
`stackn ame.em pla ce( value)` - the element is added to the stack at the top position O(1)

Queue

`empty()` - O(1)
`size()` - O(1)
`swap()` - O(1)
`emplace()` - O(1)
`front()` - return the first element of the queue O(1)
`back()` - return the last element of the queue O(1)
`push()` - O(1)
`pop()` - O(1)

HashMap & HashSet

HashMap	HashSet
`unorde red _ma p<s tring, int> umap`	`unorde red _se t<d ata _ty pe> name`

Data Strcuture C++ Cheat Sheet

by eyan via cheatography.com/197714/cs/41730/

HashMap & HashSet (cont)

at() - returns the reference to the value with the element as key k	size()
begin()	empty()
end()	find()
count()	erase()
find()	insert()
empty()	
erase()	

intSet.fi nd(arr[i]) == intSet.end()

Vector

Initialize - vector <in t> new(2)
Return return{i, j}
Sort sort(v.be gin(), v.end())
empty()
push_b ack()
emplac e_b ack()
pop_back()
reverse()
erase()

Character / Integer / Other

Character	Integer	Other
isalnum() - check if the character is a digit or a letter	to_str ing (in teg er)	isdigit()
tolower()		rand()
		lower_ bound() / upper_ bound() stoi(< str i ng >)

Sliding Window

Window Size: You have a 'window' of a fixed size, which is just a range that can look at a certain number of elements in the sequence at once.
Sliding the Window: You start with this window at the beginning of your sequence. Then, you move (or slide) this window one element at a time towards the end.
Processing Data: At each step, you perform some kind of calculation or check on the elements within the window. This could be finding the sum of numbers, checking for a pattern, and so on.
Result: The outcome of the algorithm depends on the problem you're solving.

`//add the string to hashSet or hashMap
int i = 0, j = 0;
while (j < vector.size())
if(mp.find() == mp.end()){
mp[string]++;
} else (j - i + 1) `

Binary Search

Start in the Middle: Open the book to the middle page. **Compare:** Check if the page you opened is the one you're looking for. If it is, great, you're done! If not, decide if the page you're looking for is before or after the current page. If the page number you want is higher than the middle page, you ignore all the pages before the current page. If the page number you want is lower, you ignore all the pages after the current page.
Repeat: Take the half of the book you decided to keep (based on the above step) and open to the middle page of that section. Repeat the comparison process.
Narrow Down: Each time you do this, you cut the number of pages you have to search in half. This makes finding the page much faster than flipping through each page one by one.

```
int low = 0, high = nums.size() - 1;

while (low <= high) {
int mid = (low + high) / 2;

if (nums[mid] == target) {
return mid;
}

if (nums[low] <= target && target < nums[mid]) {
high = mid - 1;
} else {
low = mid + 1;
}

if (nums[mid] < target && target <= nums[high]) {
low = mid + 1;
} else {
high = mid - 1;
}

return -1;
```

Two Pointers & Fast & Slow Pointers

Two Pointers

```
int start = 0;
int end = vector.size() - 1;
 while (start < end)...
```

Fast Slow Pointers `Node *slow_p = list, *fast_p = list;`

```
while (slow_p && fast_p && fast_p ->next) {
slow_p = slow_p->next;
fast_p = fast_p ->n ext ->next; ...
```