Linked List and Hash Table Cheat Sheet

by BearTeddy via cheatography.com/84248/cs/19911/

InsertHead LL

```
template <class T>
void List<T>::insertHead(T n)
{
ListNode<T> *aNewNode = new
ListNode<T>(n);
aNewNode->_next = _head;
_head = aNewNode;
_size++;
};
```

Remove Head LL

```
template <class T>
void List<T>::removeHead()
{
   if (_size > 0) {
      ListNode<T> *temp = _head;
      _head = _head->_next;
      delete temp;
      _size--;
    }
}
```

Iterative Print LL

```
template <class T>
void List<T>::print(bool withNL)
{
ListNode<T> *temp = _head;
   while (temp) {
        cout << temp->_item;
        if (withNL)
          cout << endl;
        else
          cout << " ";
        temp = temp->_next;
     }
     cout << endl;
}
```

Exist LL

```
template <class T>
bool List<T>::exist(T n)
{
for (ListNode<T>* ptr = _head; ptr; ptr = ptr-
>_next) {
if (ptr->_item == n)
{return true;}
}
return false;
```

ReverseOP LL

```
template <class T>
void List<T>::reverseOp() {
ListNode<T> *previous = NULL;
//start from NULL
ListNode<T> *current = _head;
ListNode<T> *next;
while (current != NULL) {
next = current->_next;
current->_next = previous;
previous = current;
current = next;
}
_head = previous;
}
```

Extract Max LL

```
template <class T>
T List<T>::extractMax()
{
if (_head != NULL) {
ListNode<T>* largest = _head;
ListNode<T>* location = _head;
T max = largest->_item; //
//allocating the node before the
largest.
for (ListNode<T>* ptr = _head;
ptr; ptr = ptr->_next) {
if (ptr->_next != NULL) {
if (ptr->_next->_item > largest-
>_item) {
largest = ptr->_next;
location = ptr;
max = largest->_item;
}
}
}
// removing and joining
if (largest == _head) {
_head = largest->_next;
delete largest;
_size--;
}
else {
location->_next = largest->-
_next;
delete largest;
_size--;
}
```

Extract Max LL (cont)

```
return max;
}
return T();
}
```

Operator Overloading

```
#include <iostream>
    using namespace std;
    class Cal {
        public:
    static int add(int a,int b){
            return a + b;
        }
    static int add(int a, int b,
int c)
        {
            return a + b + c;
        }
    };
    int main(void) {
        Cal C; // class object
declaration.
        cout<<C.add(10, 20)
<<endl;
        cout<<C.add(12, 20, 23);
       return 0;
    }
```

Hash Insert

```
hash-insert(key, data)
int i = 1; // num of collisions
while (i <= m) { // Try every bucket
int bucket = h(key, i);
if (T[bucket] == null){ // Found an empty
bucket
T[bucket] = {key, data}; // Insert key/data
return success; // Return
}
i++;
}
HandeError(); // Table full!
```

Hash Search

```
hash-search(key)
int i = 1;
while (i <= m) {
int bucket = h(key, i);
```

```
}
```

```
if (T[bucket] == null) // Empty bucket!
return key-not-found;
if (T[bucket].key == key) // Full bucket.
return T[bucket].data;
i++;
}
return key-not-found; // Exhausted entire
table
```

Quick Sort

```
int partition (int arr[], int
low, int high)
{
    int pivot = arr[high]; //
pivot
    int i = (low - 1); // Index
of smaller element

    for (int j = low; j <= high
- 1; j++)
    {
        // If current element is
smaller than or
        // equal to pivot
        if (arr[j] <= pivot)
        {
            i++; // increment
index of smaller element
```


By **BearTeddy**
cheatography.com/bearteddy/

Published 21st June, 2019.
Last updated 21st June, 2019.

Quick Sort (cont)

```
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

/* The main function that implements QuickSort
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is
now
        at right place */
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

By **BearTeddy**
cheatography.com/bearteddy/

Published 21st June, 2019.
Last updated 21st June, 2019.