Middle East Technical University | Department of Computer Engineering

CENG 140

C Programming

Spring 2016-2017

Take Home Exam 2

Due date: 10 June 2017, Saturday, 23:55

1 Objectives

In this assignment, you will implement a multi-level linked list structure.

Keywords: *Linked Lists, Search, Traversal*

2 A Multi-level Linked List

The linked list that you will implement in this assignment will keep its nodes sorted by their keys. You will insert new items in sorted order, you will also search and remove items in a given list. The list will have multiple levels and duplicate some of its nodes in upper levels. These duplicate nodes will improve search speed in the list structure by providing jump points. The duplication rule is explained in Section 2.2.

Figure 1 provides an example multi-level linked-list. Each node in the list is a `struct node` as explained in the Section 2.1.

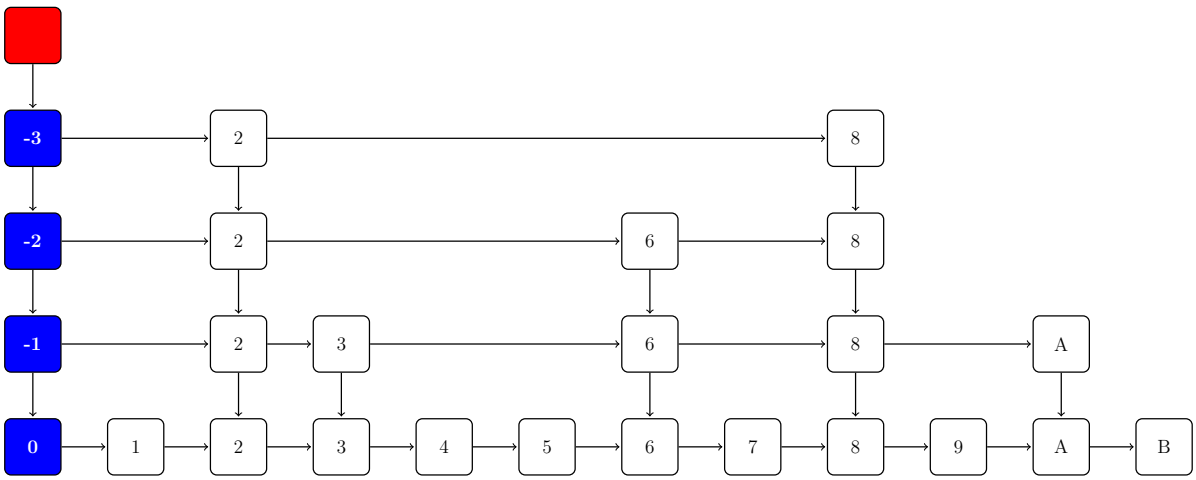

Figure 1: Linked list structure

2.1 `struct node`

The nodes of the linked list contain a *key-value* pair and two pointers: *next*, to the next node on the same level, and *down*, to the node on the level below. The key is a positive integer and it is used to sort the nodes in the list. The value is some arbitrary data and it is an array of 5 char's. All nodes in the linked list contain **unique** keys but they may have the same values.

```
typedef struct node{
    int key;
    char value[V_SIZE];
    struct node *next;
    struct node *down;
} node;
```

For convenience, we define 3 macros with the names below. These will not change your implementation, but will be used for alignments in printing the list. The keys of the nodes are 3 digit (at most) integers and therefore the list have at most 999 nodes at the lowest level. The values hold in the nodes are 5-char arrays. You do not need to check for the validity of the inputs, they will be in the range and you can use these macros in your implementations.

```
#define LIMIT 1000
#define K_SIZE 3
#define V_SIZE 5
```

2.2 List Details

The list will contain dummy nodes at each level. These nodes do not contain data (key:value pairs) inserted into the list. The red node in Figure 1 is the list head. The blue nodes are the level head nodes. These dummy head nodes do not contain any values. The keys of the heads are used to keep information about the linked list. The white nodes in Figure 1 are the data nodes that are inserted to the list.

You will be given a number when the linked list is created. We will call this the branch factor (B). This number will be kept at the list head and used in the duplication rule. The list head will also keep the number of nodes in the list (N), which is always less than 1000. Therefore the key of the list head will be $B * 1000 + N$ and it will be updated after each insert and delete operation. The level heads will keep the level number L as $-1 * L$, the lowest level that keep all inserted nodes is level 0, upper levels are numbered as $-1, -2, ..., -L$ for an $L + 1$ level linked list. Since the keys of the actual nodes are all positive integers, level head keys will always be less than a node key and will be in sorted order. This will make your linked list operations safe, you do not need to have exceptional cases for the first nodes.

The duplications to upper levels will be decided based on the number of nodes in the list. Let's say, when you insert a new node there will be N nodes in the linked list with the branch factor B, here are the rules:

- If B does not divide N, the new node will only be inserted to the lowest level.
- If some B^D divides N, the new node will be inserted to all D levels above the lowest level.

When B=2:

if N is odd, only insert to level 0
if $N = 2 (2 == 2^1)$, insert to levels 0 and 1
if $N = 6 (6\%2^1 == 0)$, insert to levels 0 and 1
if $N = 12 (12\%2^2 == 0)$, insert to levels 0, 1, 2
if $N = 4 (4 == 2^2)$, insert to levels 0, 1, 2
if $N = 8 (8 == 2^3)$, insert to levels 0, 1, 2, 3
if $N = 16 (16 == 2^4)$, insert to 0, 1, 2, 3, 4

3 Operations

The function prototypes are given in the header file *the2.h*. You will implement these functions in *the2.c*. There are many common parts in these functions, moving these parts into separate functions will reduce your work, make your solution cleaner and shorter, and finally make debugging easier. At least, try to add a common search function that you can use in insert, delete and find operations. Think a bit on paper before coding.

3.1 `node *init(int branch);`

This function will create a linked list and return a pointer to its head node. After initialization the list structure contains 2 dummy head nodes, one for the list head that contains the branch factor and the number of nodes (0 at this point) and one for the level 0 without any real data nodes.

```
node *list = init(2);
```

3.2 `void clear(node *list);`

This method will free every allocated node for this linked list. You need to remove all of the nodes in the linked list from the memory.

```
clear(list);
```

3.3 `int is_empty(node *list);`

`int num_levels(node *list);`

`int num_nodes(node *list);`

These three methods are simple queries into the linked list. The first one checks whether the list is empty or not. The list will be empty when it's created and will not be empty if there are nodes which are inserted but not deleted. Return 1 if it is empty 0 otherwise. The second one will return the number of levels in the linked list. A newly initialized list has 1 level without any nodes. The last one will return the number of nodes (level head node (dummy) is not counted) in the lowest level. This is the difference between insertions and deletions, and it is also the number you need to keep in the list head node. The sample outputs are valid for Figure 7.

```
printf("is_empty: %d\n", is_empty(list));
printf("num_levels: %d\n", num_levels(list));
printf("num_nodes: %d\n", num_nodes(list));
```

```
is_empty: 0
num_levels: 4
num_nodes: 10
```

3.4 `node *insert(node *list, int key, char *value);`

This will allocate and insert a new node to the list if *key* does not exist in the list. If the key exists, just update its value for all necessary levels. Return the pointer to the node at the lowest level on success, return NULL on failure. You may need to insert a new node into multiple levels, it will be decided based on the current number of nodes as described in Section 2.2. Do not forget to update the node count at the list head.

```
insert(list, 123, "abc");
```

3.5 `int delete(node *list, int key);`

This will remove the node with given key from the linked list. You need to delete nodes from all necessary levels. Return 1 on success, return 0 if key does not exists or in case of other errors. Do not forget to decrease the node count. Delete necessary levels that does not contain any nodes after deleting the node with given key. Do not delete level 0. If all nodes are deleted, the list will look like a newly created list.

```
delete(list, 123);
```

3.6 `void print(node *list);`

This will print the linked list structure, one line per node in the lowest level. The first line will contain the number of nodes and level ids as integers from 0 to num_levels-1. For every node we use a '+' character after a single space, except we show the list head with a '-' on the second line. The keys and values are right justified (padded with spaces) with 3 and 5-character limits respectively. They are separated with a single ':' character. The example shows the the linked list in Figure 7. Note that for every duplicate node in upper levels, we have a single '+' denoting the node.

```
print(list);
```

```
 10       0 1 2 3
          + + + + -
123:  abc +
234:  cde + +
345:  wer +
456:  rqe + + +
567:  rre +
678:  yey + +
789:  rtr +
890:  htv + + + +
901:  bgh +
912:  bnm + +
```

3.7 `void print_level(node *list, int level);`

This function will print the contents of a level with the pointers from the level above. The output contains 3 lines for all levels except the highest one which only contains a single line. The first line contains the duplicate nodes in the level above, the second line just contains '|' characters as down arrows aligned with ':' characters, the third line contains the nodes in the given level. Put a right arrow as ' -> ', with single spaces before and after, after the nodes. The example shows level 2 of the same linked list in Figure 7.

```
print_level(list, 2);
```

```
                          890:  htv ->
                              |
 -2:       -> 456:  rqe -> 890:  htv ->
```

3.8 `node *find(node *list, int key);`

This method will find and return the pointer of the node with *key*. It will return NULL if the key does not exist in the list. If the key exists in multiple levels, the returned pointer is the one

at the highest level (which has the shortest path from the list head).

Find operation starts from the list head, searches the upper most level, if its finds the key, it returns. If it finds a key value greater than the searched key, it goes one level down from the last node (the previous node with a key less than the searched key, but the greatest upto that point) and continues searching.

```
n = find(list, 234);
if (n)
    printf("Found node %d:%s at %p \n", n->key, n->value, (void*)n);
else
    printf("Cannot find node %d\n", 234);
```

```
Found node 234:cdehg at 0x97f30f8
```

3.9 `void path(node *list, int key);`

It prints keys as integers (aligned right with with spaces) and the followed paths as next:‘ > ’ or down:‘ v ’ (with a single space before and after > and v characters). Finally, it prints the key:value pair aligned with spaces (3 for keys, 5 for values). The key will be in the linked list, you do not need to check. The example below uses a new linked list printed before the path function calls.

```
print(list);
path(list, 234);
path(list, 1);
path(list, 678);
path(list, 901);
```

```
 11        0 1 2 3
           + + + + -
  1:  ilk +
 12:  bnm + +
123:abcty +
234:cdehg + + +
345: werf +
456:  rqe + +
567:   rr +
678:    y + + + +
789: rtrd +
890:  htv + +
901:bghdf +

2011 v  -3 v  -2 > 234:cdehg
2011 v  -3 v  -2 v  -1 v   0 >   1:  ilk
2011 v  -3 > 678:    y
2011 v  -3 > 678 v 678 v 678 > 890 v 890 > 901:bghdf
```

4 Regulations

1. **Programming Language:** You will use C.
   - **E**xternal libraries are not allowed.
   - **T**he source files are compiled with:

     ```
     gcc -Wall -ansi -pedantic-errors the2.c test.c -o the2
     ```

   - Do not write a *main* function in your source file.
   - Do not remove or modify any variables or functions given in the header. You can add any variables or functions in your implementation to *the2.c*.
2. **Late Submission** is not allowed.
3. **Cheating:** We have zero tolerance policy for cheating. In case of cheating, all parts involved (source(s) and receiver(s)) get zero. People involved in cheating will be punished according to the university regulations.
4. **R**emember that students of this course are bounded to code of honor and its violation is subject to severe punishment.
5. **Newsgroup:** You must follow the newsgroup (news.ceng.metu.edu.tr) for discussions and possible updates on a daily basis.

5 Submission

- Submission will be done via Moodle on `cengclass.ceng.metu.edu.tr`.
- You will submit a single file: **the2.c**
- A test environment will be ready in Moodle.
  - You can submit your source files to Moodle and test your work with a subset of evaluation inputs and outputs.
  - This will not be the actual grade you get for this assignment.
  - Additional test cases will be added after the deadline.
- There will be test cases that evaluate each function separately, you will receive partial grades if you only complete some of the functions. Try your best.

6 Examples

You can see the structure of the generated linked list for the following insert operations in the figures below:

```
node* list = init(2);
insert(list, 123, "abc");
insert(list, 234, "cde");
insert(list, 345, "wer");
insert(list, 456, "rqe");
insert(list, 567, "rre");
insert(list, 678, "yey");
insert(list, 789, "rtr");
insert(list, 890, "htv");
insert(list, 901, "bgh");
insert(list, 912, "bnm");
print(list);
```

Expected output:

```
 10        0 1 2 3
           + + + + -
123:  abc  +
234:  cde  + +
345:  wer  +
456:  rqe  + + +
567:  rre  +
678:  yey  + +
789:  rtr  +
890:  htv  + + + +
901:  bgh  +
912:  bnm  + +
```

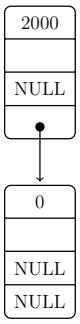

Figure 2: Empty list after init

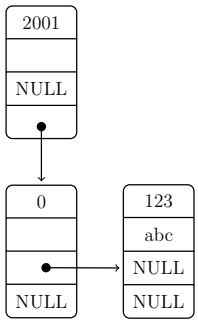

Figure 3: After inserting the first item

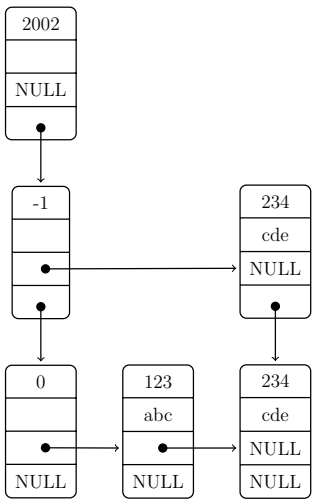

Figure 4: After inserting the second item

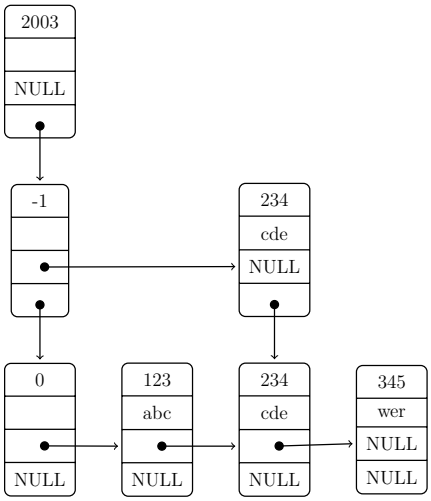

Figure 5: After inserting the third item

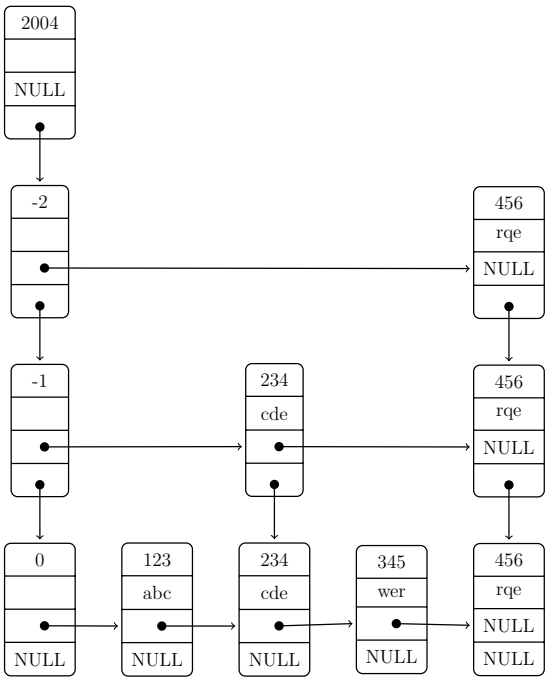

Figure 6: After inserting the fourth item to the list

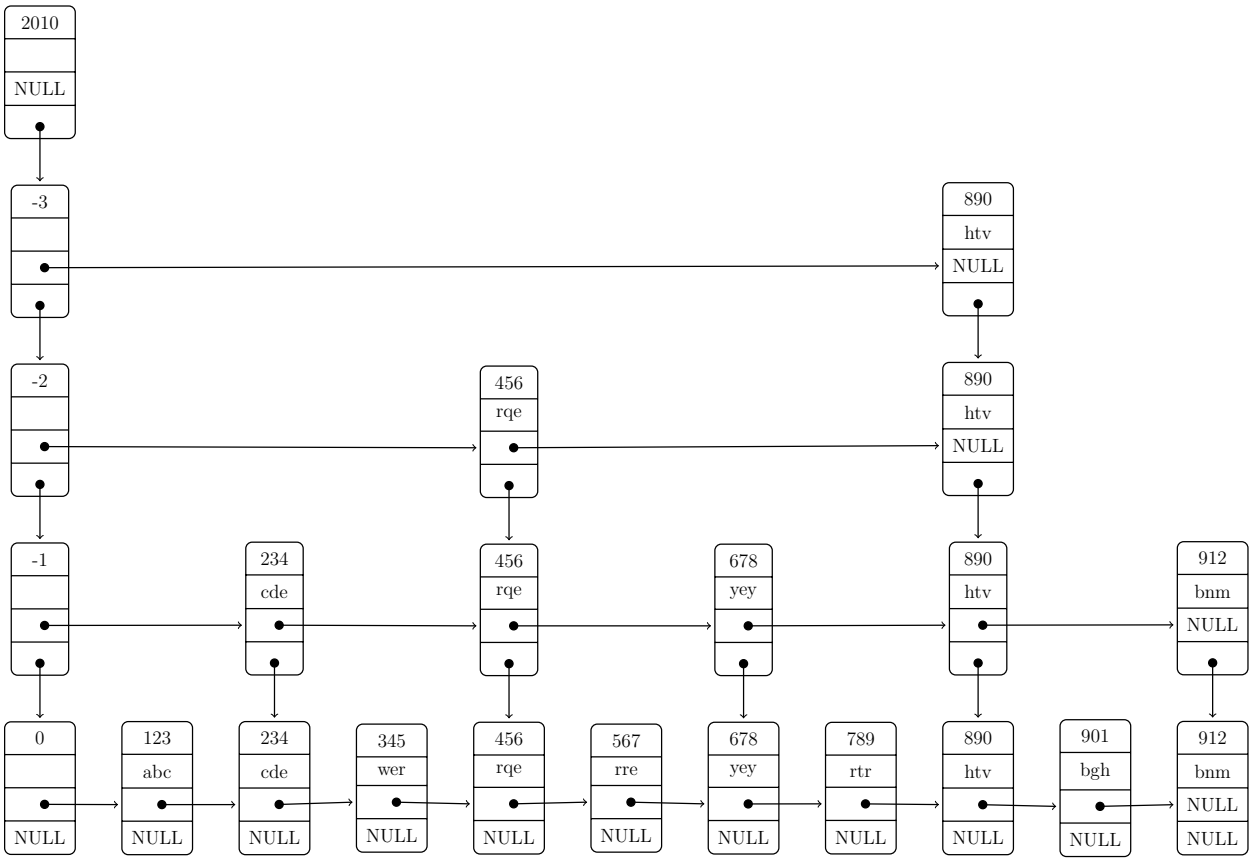

Figure 7: The list after inserting all ten items

The same list and several delete operations:

```
delete(list, 123);
delete(list, 234);
delete(list, 890);
delete(list, 912);
print(list);
```

Expected output:

```
  6        0 1 2
           + + + -
345:  wer +
456:  rqe + + +
567:  rre +
678:  yey + +
789:  rtr +
901:  bgh +
```

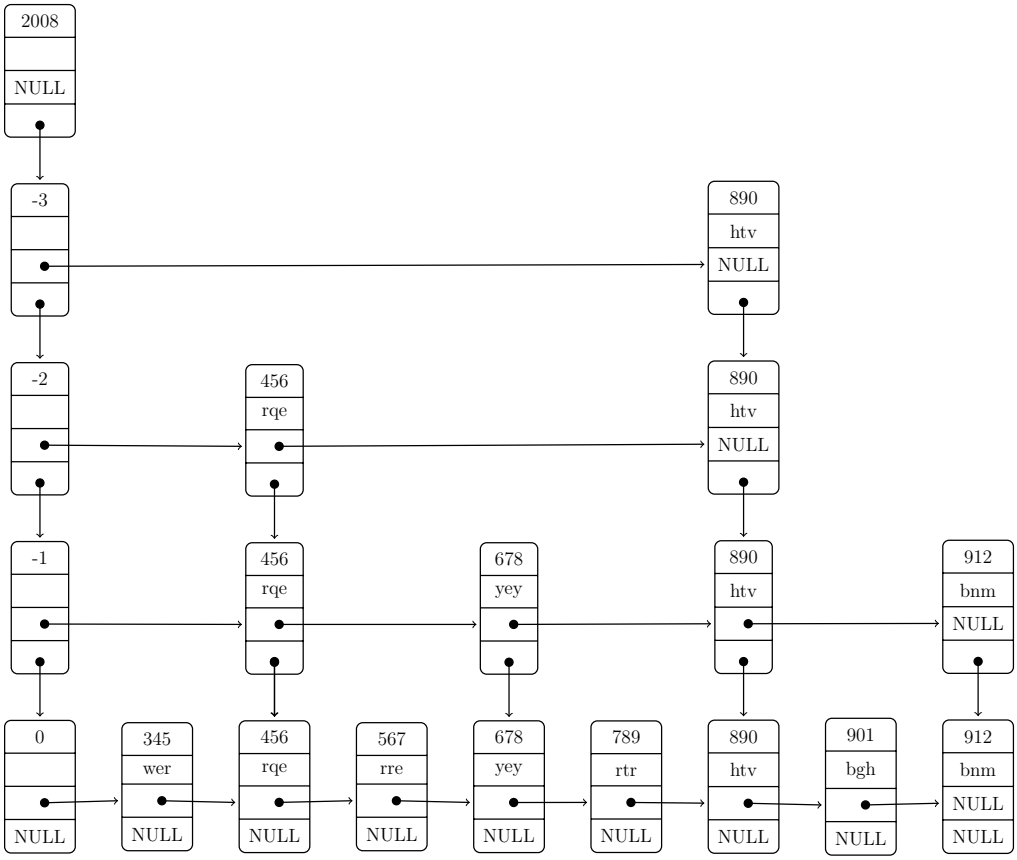

Figure 8: The list after deleting 123 and 234

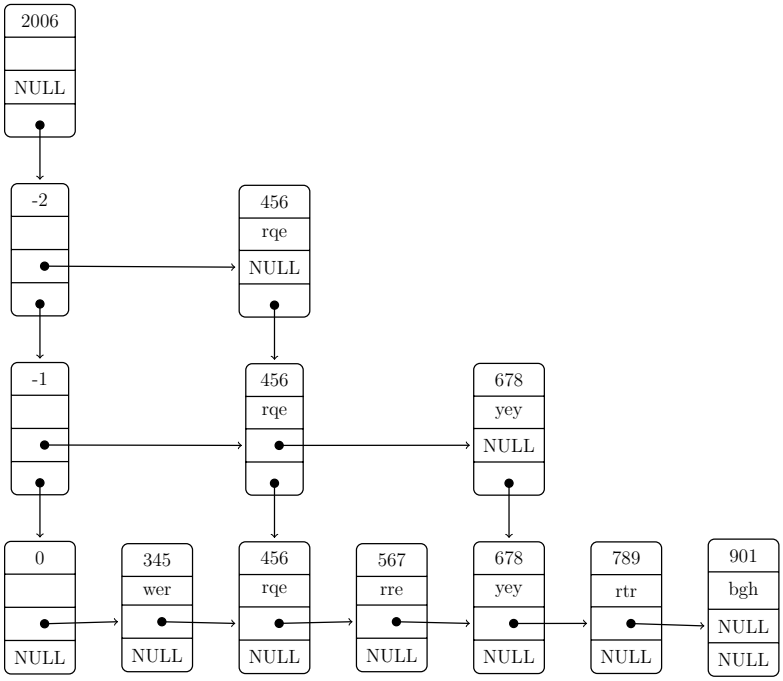

Figure 9: The list after deleting 890 and 912

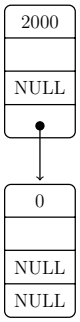

Figure 10: The final list if all items are deleted