ECE 264 Advanced C Programming

1 Doubly Linked List

Each node in our linked lists has only one link to the next node. In such a list, we can reach any node from the first node but we cannot reach the first node from any other node. We can add another link to the **previous** node and create a *doubly linked list*. Any node can reach any other node using the previous or the next link. When handling insertion and deletion, we need to make sure both the previous and the next links are updated correctly. The following code shows how to insert a value into a doubly linked list.

```
#ifndef DLINKNODE_H
#define DLINKNODE_H
typedef struct dlinknode
{
  struct dlinknode * ln_next;
  struct dlinknode * ln_prev;
  int ln_value;
} Node;

Node * DList_copy(Node * n);
void DList_assign(Node * * n1, Node * n2);
void DList_insert(Node * * n, int v);
int DList_delete(Node * * list, int v);
void DList_print(Node * n);
void DList_destruct(Node * n);
int DList_search(Node * list, int v);
#endif
```

```
#include "dlinknode.h"
#include <stdio.h>
#include <stdlib.h>
static Node * Node_construct(int v)
{
  Node * n = malloc(sizeof(Node));
  n -> ln_value = v;
  n -> ln_next = 0;
  n -> ln_prev = 0;
  return n;
}
```

```
void DList_insert(Node * * n, int v)
{
  Node * p = Node_construct(v);
  Node * curr = * n;
  Node * prev = * n;
  if (( *n) == 0)
    {
      *n = p;
      return;
    }
  while ((curr != 0) && ((curr -> ln_value) < v))
    {
      prev = curr;
      curr = curr -> ln_next;
    }
  if (curr == (* n))
    {
      p -> ln_next = (* n);
      (* n) -> ln_prev = p;
      * n = p;
    }
  else
    {
      p -> ln_next = prev -> ln_next;
      p -> ln_prev = prev;
      /* Will p be the last node in the list? */
      if ((prev -> ln_next) != 0)
        { (prev -> ln_next) -> ln_prev = p; }
      prev -> ln_next = p;
    }
}
```

2 Divide and Conquer and Dynamic Structures

When we explained binary search, we used an array. Arrays have one major advantage: we can access any element in a single step. In contrast, in a linked list, we have to follow the links node by node. However, arrays have one major restriction: the sizes are fixed. In contrast, a linked list can expand or shrink as needed. Can we combine the performance of binary search without being restricted to arrays? A *binary search tree* is one solution.

Doubly linked list is *linear* meaning that all nodes are linked as a line. If each node has two links, the links do **not** have to be related. It is **not** necessary that any node can reach any other node using the previous or the next link. It is all right if there is only one path to any node, from one special node called *root*. We usually draw a "tree" upside down. The "root" is at the top and the "leaves" are at the bottom.

3 Integer Partition

Partition an integer n into the sums of positive integers. For example, $n = 4$ can be partitioned to 4 = 3 + 1 = 2 + 2 = 1 + 1 + 2.

1. The orders are considered, 1 + 1 + 1 + 2 = 1 + 1 + 2 + 1 are two different partitions. Similarly, 1 + 3 = 3 + 1 are different.
2. The orders of the numbers are **not** considered, i.e. no permutation. Hence, 1 + 1 + 1 + 2 = 1 + 1 + 2 + 1 are the same and should **not** repeat. Similarly, 1 + 3 = 3 + 1 are the same.
3. The numbers must be distinct and increasing. For example, 1 + 3 is accepted but 2 + 2 is not accepted. Nor is 3 + 1 accepted.
4. For number n, how many ways can it be partitioned based on one of the three rules? List the answers for n between 1 and 10. Can you derive general formulas? In all cases, the number n itself is included as one partition.

Write a program that can partition any positive number n and follow one of these different rules. The output for $n = 5$ and the first rule will be in this format:

```
[1, 1, 1, 1, 1]
[1, 1, 1, 2]
[1, 1, 2, 1]
[1, 1, 3]
[1, 2, 1, 1]
[1, 2, 2]
[1, 3, 1]
[1, 4]
[2, 1, 1, 1]
[2, 1, 2]
[2, 2, 1]
```

```
[2, 3]
[3, 1, 1]
[3, 2]
[4, 1]
[5]
```