CptS 122

Dyanamic Self Referential Lists in C

Objectives

Use and explain dynamic allocation of memory in a C program.

List, describe and use the standard C library functions for dynamic memory allocation

- malloc()
- calloc()
- realloc()
- free()

Implement the List ADT as an array

Implement the List ADT as a dynamically linked data structure.

Review Dynamic Allocation for C Variables

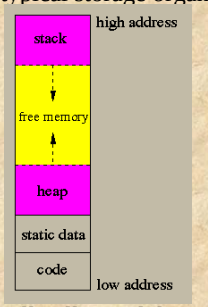
The C standard library (stdlib.h) provides three functions for dynamically allocating variables: malloc, calloc, and realloc.

All of these functions return pointers to new variables or arrays of variables, unless the system runs out of memory. In that case they return NULL.

It is a good habit to check the returned value. If it is NULL then your program should just print an error message and exit.

Storage Allocation

Consider a typical storage organization of a program:



All dynamically allocated data are stored in the heap. These are the data created by malloc in C or new in C++. You can imagine the heap as a vector of bytes (characters) and end_of_heap a pointer to the first available byte in the heap:

void *malloc(size_t number_of_bytes)

```
char *cp;

cp = malloc(100);
```

attempts to get 100 bytes and assigns the start address to cp.

Also it is usual to use the **sizeof()** function to specify the number of bytes:

```
int *ip ;
  ip = (int *) malloc( 100 * sizeof(int) );
```

Explicit Type Cast

calloc() and realloc()

There are two additional memory allocation functions, Calloc() and Realloc(). Their prototypes are given below:

```
void *calloc(size_t num_elements, size_t
element_size};
void *realloc( void *ptr, size_t new_size);
```

Malloc does not initialize memory (to zero) in any way. If you wish to initialize memory then use calloc(). calloc() is slightly more computationally expensive but more convenient than malloc.

Also note the different syntax between calloc and malloc in that calloc takes the number of desired elements, num_elements, and element_size, as two individual arguments.

realloc()

realloc() is a function which attempts to change the size of a previous allocated block of memory.

The new size can be larger or smaller.

If the block is made larger then the old contents remain unchanged and memory is added to the end of the block.

If the size is made smaller then the remaining contents are unchanged.

realloc()

If the original block size cannot be resized then realloc() will attempt to assign a new block of memory and will copy the old block contents. Note a new pointer (of different value) will consequently be returned. You must use this new value.

If new memory cannot be reallocated then realloc returns NULL.

realloc()

Thus to change the size of memory allocated to the *ip pointer from before to an array block of 50 integers instead of 100, simply do:

```
ip = (int *) realloc( ip, 50);
```


free()

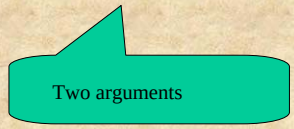
When you have finished using a portion of memory you should always free() it. This allows the memory freed to be aavailable again, possibly for further malloc() calls

The function **free()** takes a pointer as an argument and frees the memory to which the pointer refers.

calloc()

Thus to assign 100 integer elements that are all initially zero you would do:

```
int *ip;
     ip = (int *) calloc( 100, sizeof(int) );
```



A dynamic self referential list example

create an new element for the list
insert at the front of the list
insert at the end of the list
insert an element into an ordered list
remove from the front of the list
remove from the end of the list
remove anywhere in the list
return an element from the front of the list
return an element from the end of the list
print the list

Building a car structure


```
struct listNode
{
   char             dataItem[SIZE];
   struct listNode*  next;
};

typedef struct listNode ListNode;

typedef ListNode* ListNodePtr;
```


create an new element for the list


```
ListNodePtr makeNode(char* data)
{
   ListNodePtr tmp = NULL;
   tmp = (ListNodePtr)
                malloc( sizeof( ListNode ) );
   tmp->next = NULL;
   strcpy(tmp->dataItem, data);

   return tmp;
}
```

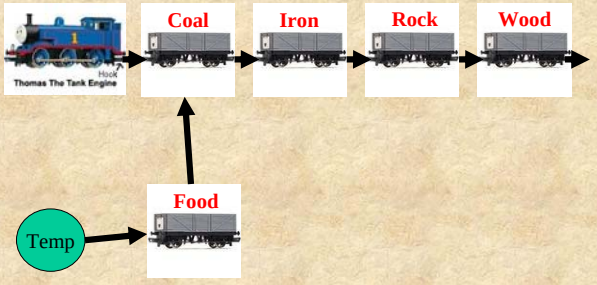

insertFront(&Thomas, Food)

insertFront(&Thomas, Food)

Create the car and load it.

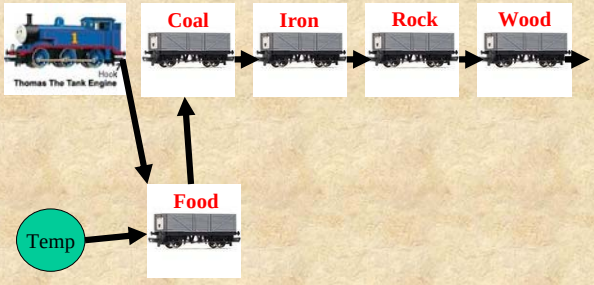
insertFront(&Thomas, Food)

Hook the car to the first car in Thomas' train



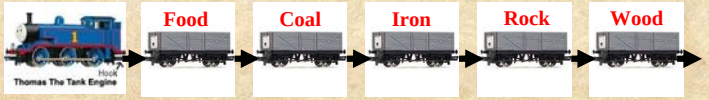
insertFront(&Thomas, Food)

Hook Thomas to the new car



insertFront(&Thomas, Food)

complete!



insertFront(&Thomas, Food)

```
void insertFront(ListNodePtr* head, ListNodePtr newNode)
{
  newNode->next = *head;
  *head = newNode;
}
```

The test driver for insertFront

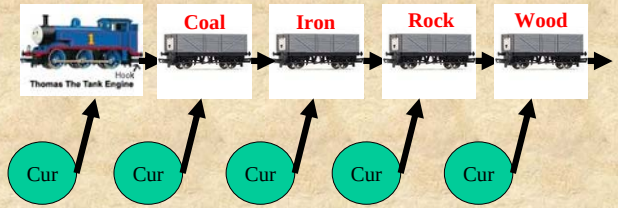
```
void main( void )
{
  ListNodePtr node = NULL;  // temp pointer for each new node.
  ListNodePtr firstList = NULL;  // the root pointer to the list

  node = makeNode("Andy");
  insertFront(&firstList, node);
  printList(firstList);
  node = makeNode("Tammy");
  insertFront(&firstList, node);
  printList(firstList);
```


insertBack(&Thomas, Food)

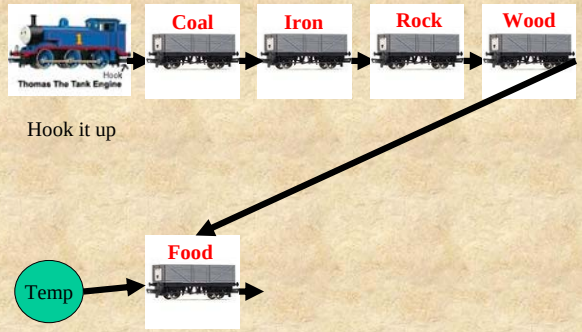
Create the car and load it

insertBack(&Thomas, Food)

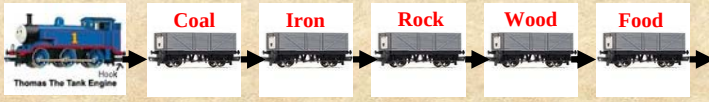


Find the end of the list

insertBack(&Thomas, Food)



insertBack(&Thomas, Food)



And it is done


```
void insertBack(ListNodePtr* head, ListNodePtr newNode)
{
   ListNodePtr current = *head ;
   if ( NULL == current ) // the list is empty
   {
      *head = newNode;
   }
   else   // the list is NOT empty, walk to the end.
   {
      while ( NULL != current->next )
      {
         current = current->next;
      }
      current->next = newNode;
   }
}
```


insertInOrder (&Thomas, Food)

1) Find the right place
2) Connect the new car to the rest of the list
3) Break the list and connect to the new car

Unfortunately there is a little more

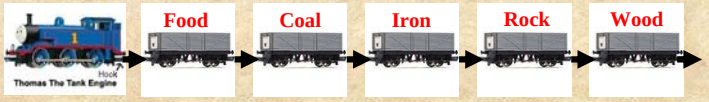
insertInOrder (&Thomas, Food)

```
void insertInOrder(ListNodePtr* head, ListNodePtr newNode)
{
  ListNodePtr current  = *head;
  ListNodePtr previous = NULL;
  if ( NULL == current )   // the list is empty
  {
    *head = newNode;
  }
  else if ( 0 < strcmp( current->dataItem, newNode->dataItem ) )
  {
    newNode->next  = current;
    *head          = newNode;
  }
```

insertInOrder (&Thomas, Food) (part 2)

```
else       // find the right place and put it in.
  {
    previous = current;
    while ( (NULL != current->next)  &&
      (0 >= strcmp( current->dataItem, newNode->dataItem ))   )
    {
      previous = current;
      current  = current->next;
    }
    insertFront( &(previous->next) , newNode);
  }
}
```


printList (Thomas)



printList (Thomas)

```
void printList(ListNodePtr list)
{
   while ( NULL != list )
   {
      printf("%s ==> ", list->dataItem);
      list = list->next;
   }
   printf(" NULL \n");
}
```


removeFront(&Thomas)

```
void removeFront(ListNodePtr* head)
{
   ListNodePtr tmp = *head;
   if ( NULL != tmp )
   {
      *head = (*head)->next;
      free( tmp );
   }
}
```

removeBack(&Thomas)

```
void removeBack(ListNodePtr* head)
{
   ListNodePtr current = *head;
   ListNodePtr previous = *head;
   if ( NULL != *head )
   {
      if ( NULL == (*head)->next )
      {
         free ( *head );
         *head = NULL;
      }
```

removeBack(&Thomas)

```
else
      {
         while ( NULL != current->next )
         {
            previous = current;
            current = current->next;
         }
         free (current);
         previous->next = NULL;
      }
   }
}
```

removeItem(&Thomas, “Iron”)

```
void removeItem(ListNodePtr* head, char *item)
{
   ListNodePtr current  = *head;
   ListNodePtr previous = *head;
   ListNodePtr tmp      = *head;

   if ( NULL != previous )  // the list is not empty
   {
      // the first item should be removed.
      if ( 0 == strcmp( current->dataItem, item ))
      {
         *head = (*head)->next;
         free (tmp);
      }
```

removeItem(&Thomas, “Iron”)

```
else  // maybe later in the list find it.
      {

         while ( (NULL != current)  &&
            (0 != strcmp( current->dataItem, item))   )
         {
            previous = current;
            current  = current->next;
         }
         if ( NULL != current )
         {
            previous->next = current->next;
            free (current);
         }
      }
   }
}
```