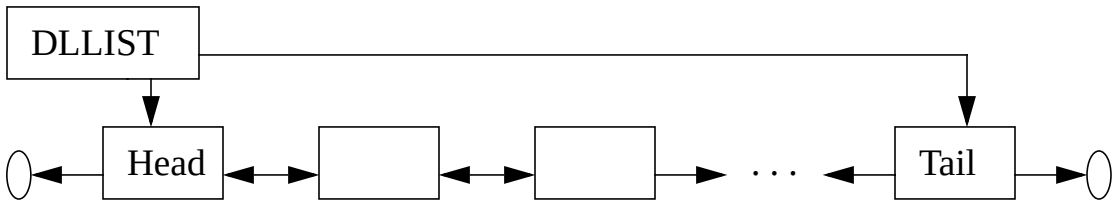
Trick ADT Library Programming Guide

The library consists of routines to perform operations on several abstract data types. Currently the library has support for linked-lists, stacks, queues, binary trees, and a string to pointer hash table.

How to read this document

This document contains sections on the Abstract Data Types (ADTs) contained in the Trick ADT library. Included with each data structure description is a graphical respresentation of the structure, function descriptions, and examples on how to use those functions are in the appendix. The function descriptions contain four sub-sections, return value, parameters, purpose, and CPU time. The CPU time representation is given in the big-O notation for worst case, average case, and best case.

Linked List



Include:
#include “tricked/dllist.h”

The DLLIST structure contains the information needed to reference a linked list. The count is the number of elements currently in the list. The head and tail pointers are given as doubly-linked list node pointers. An optional compare function pointer can be set for doing automatic list searching using the DLL_FindPos and DLL_Find functions

```
typedef struct _DLLIST
{
  int count;
  DLLNODE* head;
  DLLNODE* tail;
  int (*compare)(void* info1,void* info2);
}DLLIST;
```

DLLIST* DLL_Create()

Return Value

returns a pointer to an empty, initialized linked list handle.

Purpose

Always use DLL_Create when dynamically creating a linked list. DLL_Delete is used to release the memory allocated by DLL_Create. For an example, see example A.1

CPU Time

Average: O(1)
Best: O(1)
Worst: O(1)

void DLL_Delete(DLLIST * pList)

Return Value

none

Parameters

pList: Pointer to the doubly-linked list to be deleted

Purpose

Use DLL_Delete to remove all elements from a linked list, and free memory allocated by the linked list functions. User data stored in the list is not freed. After calling DLL_Delete, the list pointed to by pList is unusable because the memory has been freed.

CPU Time

Average: O(n)
Best: O(n)
Worst: O(n)

void DLL_Init(DLLIST * pList)

Return Value:

none

Parameters:

pointer to the linked-list that is to be initialized

Purpose

This function can be used to initialize a statically allocated linked list, or to initialize a DLLIST structure that was allocated with malloc by the user.

CPU Time

Average: O(1)
Best: O(1)
Worst: O(1)

int DLL_GetCount(DLLIST * pList)

Return Value

number of elements in the list

Parameters:

pointer to the linked-list

Purpose

This function can be used to determine how many elements are in the list

CPU Time

Average: O(1)
Best: O(1)
Worst: O(1)

void * DLL_Find(void * data,DLLIST * pList)

Return Value

pointer to the element that matches data., or NULL if data not found

Parameters:

data: pointer to the data to be found in the list

Purpose

If you have defined a compare function for this list you can use DLL_Find to search for an element that contains or matches the data parameter. The criteria for a match is determined by the user defined compare function

CPU Time

Average: O(n)
Best: O(1)
Worst: O(n)

DLLPOS DLL_FindIndex(int index,DLLIST * pList)

Return Value

Position handle of the position at index.

Parameters:

index: zero-based element from the head of the list. index 0 corresponds to the head, index 1 corresponds to the element immediatly after the head. The value of index should not exceed the value of DLL_GetCount() - 1.

pList: pointer to the list

Purpose

Use this function when you want to access an element of the list like you would with an array, where index is the array subscript. To get each element in the list in a particular order, use DLL_Next / DLL_GetPrev because they are much more efficient than DLL_FindIndex

CPU Time

Average: O(1)
Best: O(1)
Worst: O(1)

DLLPOS DLL_FindPos(void * data,DLLIST * pList)

Return Value

position of the element that matches data, or NULL if a match is not found

Parameters:

data: pointer to the data to be found in the list

Purpose

If you have defined a compare function for this list you can use DLL_FindPos to search for an element that contains or matches the data parameter. The criteria for a match is determined by the user defined compare function

CPU Time

Average: O(n)
Best: O(1)
Worst: O(n)

void * DLL_GetAt(DLLPOS pos, DLLIST * pList)

Return Value

pointer to the element that is located at the list position pos

Parameters:

pos: list position that contains the data to be returned

Purpose

gets the data that is stored at a position in the list. To get a position handle, use DLL_GetHeadPosition, DLL_GetTailPosition, and DLL_FindIndex

CPU Time

Average: O(1)
Best: O(1)
Worst: O(1)

void * DLL_SetAt(DLLPOS pos, void * data, DLLIST * pList)

Return Value

pointer to the data the was previously at this position

Parameters:

data: pointer to the data to be put at this position
pos: position in the list where the data is to go

Purpose

Use DLL_SetAt to change the data that is stored at an already existing position in the list.

CPU Time

Average: O(1)
Best: O(1)
Worst: O(1)

void * DLL_RemoveAt(DLLPOS pos, DLLIST * pList)

Return Value

pointer to the data stored at the list position pos

Parameters:

pos: position in the list that is to be removed

Purpose

Use DLL_RemoveAt when you want to remove an element from the list, and the position of the element is known, or has been found. see example A.3

CPU Time

Average: O(1)
Best: O(1)
Worst: O(1)

void DLL_RemoveAll(DLLIST * pList)

Return Value

none

Parameters:

pList: pointer to the list

Purpose

Use this function to remove all the elements from a list. This function does not free the user data that is stored in the list, just the positions.

CPU Time

Average: O(1)
Best: O(1)
Worst: O(1)

DLLPOS DLL_InsertBefore(DLLPOS pos, void * data, DLLIST * pList)

Return Value

the list position of the element that was inserted

Parameters:

pos: the list position that is immediatly after the insertion point in the list
data: the user data the is to be inserted into the list

Purpose

Use this function to insert an element immediatly before a known list position. A typical application of this function could be if elements are to be inserted into a list that must have the elements arranged in some order e.g. alphabetical, numerical, etc.

CPU Time

Average: O(1)
Best: O(1)
Worst: O(1)

DLLPOS DLL_InsertAfter(DLLPOS pos, void * data, DLLIST * pList)

Return Value

The list position of the element that was inserted

Parameters:

pos: the list position that is immediatly before the insertion point in the list
data: the user data the is to be inserted into the list

Purpose

Use this function to insert an element immediatly after a known list position. A typical application of this function could be if elements are to be inserted into a list that must have the elements arranged in some order e.g. alphabetical, numerical, etc.

CPU Time

Average: O(1)
Best: O(1)
Worst: O(1)

void * DLL_GetNext(DLLPOS * pos,DLLIST * pList)

Return Value

pointer to the data that is currently stored at position pos

Parameters:

pos: pointer to the position handle from which the next position is to be obtained.
Note: This function will change the position handle pointed to by pos to reference the position following the current position.

Purpose

Use this function to get the position of the element after the position that pos points to. This function is particularly useful when you need to iterate through the elements in a list from the first element to the last element. See the example A.2 for an example of using DLL_GetNext

CPU Time

Average: O(1)
Best: O(1)
Worst: O(1)

void * DLL_GetPrev(DLLPOS * pos,DLLIST * pList)

Return Value

pointer to the data that is currently stored at position pos

Parameters:

pos: pointer to the position handle from which the previous position is to be obtained.
Note: This function will change the position handle pointed to by pos to reference the position before the current position.

Purpose

Use this function to get the position of the element before the position that pos points to. This function is particularly useful when you need to iterate through the elements in a list from the last element to the first element.

CPU Time

Average: O(1)
Best: O(1)
Worst: O(1)

DLLPOS DLL_AddHead(void * data,DLLIST * pList)

Return Value

list position of data

Parameters:

data: pointer to data that is to be inserted at the head of the list

Purpose

Use this function to insert an element at the head of the list.

CPU Time

Average: O(1)
Best: O(1)
Worst: O(1)

DLLPOS DLL_AddTail(void * data,DLLIST * pList)

Return Value

list position of data

Parameters:

data: pointer to data that is to be inserted at the tail of the list

Purpose

Use this function to insert an element at the tail of the list.

CPU Time

Average: O(1)
Best: O(1)
Worst: O(1)

DLLPOS DLL_GetHeadPosition(DLLIST * pList)

Return Value

the list position of the head of the list, or NULL if list is empty

Parameters:

pList: pointer to the list

Purpose

Use this function to get the head position of the linked list

CPU Time

Average: O(1)
Best: O(1)
Worst: O(1)

DLLPOS DLL_GetTailPosition(DLLIST * pList)

Return Value

the list position of the tail of the list, or NULL if list is empty

Parameters:

pList: pointer to the list

Purpose

Use this function to get the tail position of the linked list

CPU Time

Average: O(1)
Best: O(1)
Worst: O(1)

Appendix A - Examples

A.1 How to create and insert an element into a linked list

```
#include “trick_adt/dllist.h”
#include <stdlib.h>

void main()
{
  DLLIST * mylist;
  void * mydata;

  mylist = DLL_Create();
  mydata=malloc(10);
  DLL_AddTail(mydata,mylist);

  . . . . .
  . . . . .

  free(mydata);
  DLL_Delete(mylist);


}
```

A.2 How to navigate through a list using GetNext

```
#include “trick_adt/dllist.h”
#include <stdlib.h>

void main()
{
  DLLIST * mylist;
  DLLPOS pos;
  void * mydata;

  . . . . .    /* code to add data to the list */
  . . . . .

  pos=DLL_GetHeadPosition(mylist);
  while(pos != NULL)
  {
    mydata=DLL_GetNext(&pos,mylist);
    myfunction(mydata);
```

```
  }

  . . . . .
  . . . . .
}
```

A.3 How to search for and remove an element from the list

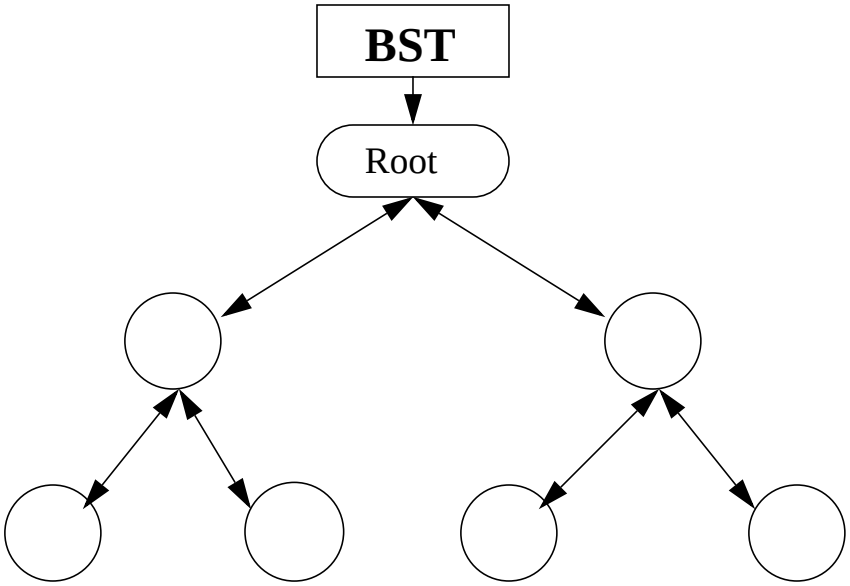
```
#include “trick_adt/dllist.h”
#include <stdlib.h>

void main()
{
  DLLIST * mylist;
  DLLPOS pos,prevpos;
  void * mydata;

  . . . . .    /* code to add data to the list */
  . . . . .


  pos=DLL_GetHeadPosition();
  while(pos!=NULL)
  {
    prevpos=pos;
    mydata=DLL_GetNext(pos,mylist);
    if(IsCorrectData(mydata))
    {
      DLL_RemoveAt(prevpos,mylist);
      free(mydata);
      break;
    }
  }
}
```

Binary Search Tree



The binary search tree is a non-linear data structure designed to provide quick access to unsorted data. The scheme uses a user-defined key to arrange the elements of data in such a way as to make searching for a particular piece of data much quicker than a linear search through a list or array. In order to use this structure, the user must determine what field in their data type they will use as the key. Example:

```
typedef struct _Customer
{
  int id;
  char * name;
}Customer;
```

If you wanted to quickly find a customer from a group of customers based on the customer id, you would use the id field as the key, and store your customer records in a binary tree. On the other hand, if in most cases you had the customer name, but needed to look up the id, then name wuold be used as the key. You would be responsible for writing a compare function that would take two customer records and return -1 if record 1 is less than record 2, or 1 if record 1 is greater than record 2, or 0 if they are equal. If the customer id is the key, the compare function could be implemented like this:

```
int compare(void* rec1, void* rec2)
{
  Customer* pCust1=(Customer*)rec1;
  Customer* pCust2=(Customer*)rec2;
  if(pCust1->id - pCust2->id < 0)
    return -1;
  else if(pCust1->id - pCust2->id > 0)
    return 1;
  else
    return 0;
}
```

If name was your key, you could use the return value from strcmp to determine if one name is less than the other.

The structure used to hold a reference to the binary tree consists of a pointer to the root node, a count of the nodes, and functions pointers. You are only responsible for setting the compare function pointer. Use the bstSetCompareFunc to set the compare function.

```
typedef struct
{
   bstNode *root;
   int nodes;
   short deltype;
   int (*compare)(void *left,void *right);
   void (*print_node)(FILE *output,void *info);
   void (*malloc_fail_handler)(void);
   #ifndef NDEBUG
      short init;
   #endif
}BST;
```

bstNode * bstFind(void * info, BST * bst)

Return Value

the node that matches info, or NULL if info not found

Parameters:

info: pointer to the data containing the key being searched for
bst: pointer to the binary search tree to be searched

Purpose

Use this function to find the node that matches the data pointed to by info. The bstFind function compares info to the info stored at nodes in the tree as it searches. The user defined compare function is called to make the comparison, with the two parameters sent to compare being info, and the data stored at the current node in the traversal. Therefore, it is important that info consist of some data type that the user defined compare function knows how to handle

CPU Time

Average: $O(Log_2 n)$
Best: O(1)
Worst: O(n)

void * bstGetInfo(bstNode * node)

Return Value

The user data that is stored at this node

Parameters:

node: points to the node that contains the user data to be returned

Purpose

Use this function to get the user data from a node in the binary search tree. Typically, this function is used after a call to bstFind

CPU Time

Average: O(1)
Best: O(1)
Worst: O(1)

bstNode * bstGetLeft(bstNode * node)

Return Value

the node that branches to the left off of this node, or NULL if there isn't a left branch

Parameters:

node: The node from which the left branch is to be obtained

Purpose

Use this function to manual traverse the tree from a particular starting node

CPU Time

Average: O(1)
Best: O(1)
Worst: O(1)

bstNode * bstGetRight(bstNode * node)

Return Value

the node that branches to the right off of this node, or NULL if there isn't a right branch

Parameters:

node: The node from which the right branch is to be obtained

Purpose

Use this function to manual traverse the tree from a particular starting node

CPU Time

Average: O(1)
Best: O(1)
Worst: O(1)

bstNode * bstGetParent(bstNode * node)

Return Value

the node that is the parent of this node, or NULL if this node is the root node

Parameters:

node: The node from which the parent is to be obtained

Purpose

Use this function to manual traverse the tree from a particular starting node, or to determine if a particular node is the root node

CPU Time

Average: O(1)
Best: O(1)
Worst: O(1)

void * bstDelete(bstNode * node, BST * bst)

Return Value

returns the data stored at node

Parameters:

node: The node to be deleted.
bst : pointer to the BST structure that references the tree containing node

Purpose

Use this function to delete a node from the binary tree. The delete strategy will use an alternating leaf replacement scheme in an attempt to keep the tree balanced during multiple deletes.

CPU Time

Average: $O(Log_2(n))$
Best: $O(Log_2(n))$
Worst: $O(n)$

int bstGetCount(BST * bst)

Return Value

returns the number of nodes in the binary tree

Parameters:

bst : pointer to the BST structure that references the tree from which the count is returned

Purpose

Use this function to obtain a count of the number of nodes in the tree

CPU Time

Average: O(1)
Best: O(1)
Worst: O(1)

bstNode * bstInsert(void * info, BST * bst)

Return Value

returns the node where info is stored

Parameters:

info: the data that is to be inserted into the tree
bst : pointer to the BST structure that reference the tree that info is to be inserted into

Purpose

Use this function to insert data into the tree. It is very important that the data being inserted is not sorted based on its key field. If sorted data is inserted into the tree, the performance for bstFind, bstInsert and bstDelete will be at worst case. Inserting pre-sorted data effectively makes the BST a linked-list.

CPU Time

Average: $O(Log_2(n))$
Best: O(1)
Worst: O(n)

void bstInit(BST * bst)

Return Value

none

Parameters:

bst: pointer to the BST structure to be initialized

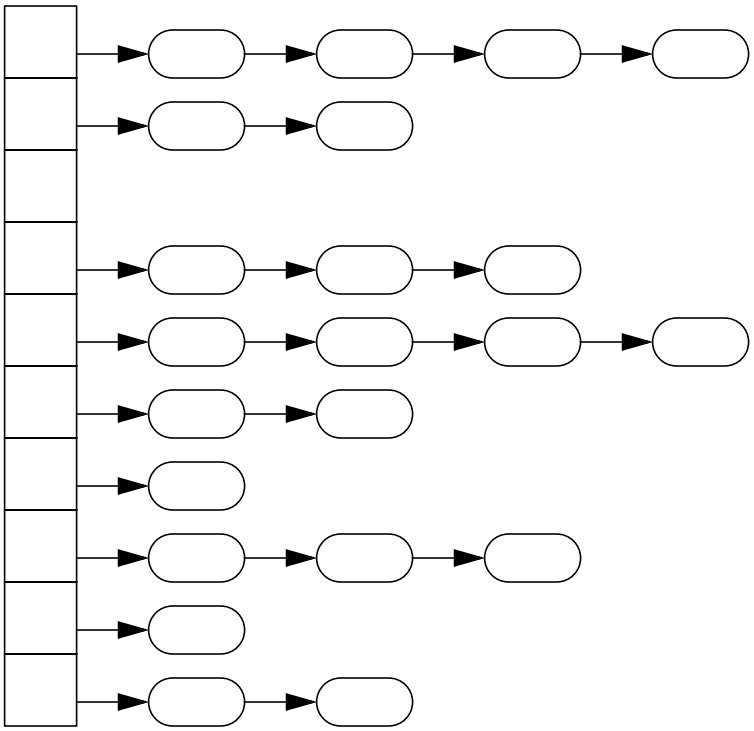
Purpose

Use this function to initialize a BST structure after it has been allocated

CPU Time

Average: O(1)
Best: O(1)
Worst: O(1)

MapStrToPtr



This data structure uses a hash table to map strings to pointers. Common uses of this type of structure could be to store the address of named references. This could be using a filename to reference the data stored in a file that has been previously read in. Essentially, any piece of data that needs to be associated with a string, and then accessed using this string can be efficiently accessed using this data structure. The string map is analogous to the way numbers are used to access elements of an array by using the number as an index like data=myarray[number]; This data structure gives you the ability to access data in a similar way: data=SM_Map("mydata",mymap); The advantage of this abstract data type over other types like arrays is, as elements are added to the map, the time required to retrieve an element from the map will increase much less than O(n) as in the case of can array. An alternative data structure that may give better performance in some circumstances is the binary tree. The underlying structure of the MapStrToPtr ADT is a table (array) of linked lists. In entry in the table corresponds to a hash value from 0 to the size of the array - 1. When an item is to be inserted into the map, a hash value is computed based on the key of the data being inserted. This hash value which must be from 0 to table size - 1 represents the index in the table where the data has been mapped to. The data is then inserted into the list at that table index. Retrieving the data involves computing the hash value of the key being searched for, then the list

at the hashed table index is searched for the element the fully matches the key. Because the hash function generates a number from a key, there is a chance that multiple keys will have the same hash value. When two items with the same hash value are inserted, a collision takes place. Collision resolution is done by using a linked list to store multiple items at a particular hash value. Item lookup performance decreases as more collisions occur, because the linked lists must be searched after the table lookup takes place. An optimal hash table, where no collisions have taken place, has an O(1) lookup time to find an item, but a worst case hash table, where all items inserted collide, has an O(n) lookup time. Insertion time is always O(1), and Removal time is the same as lookup time.