Data Structure Assignment #1

German language and literature 2016130927

Park Jun-Yeong

Problem #1-1

Set: A collection of 1). finite distinct elements, which could be number, name, people and so on, 2). with no particular order. Especially, a set with no elements is defined as an empty set.

In this ADT, Set is implemented by dynamically linked list so there is no in insertion failure due to lack of storage places. If there is no duplicate element in set in advance, then new element must be surely inserted. For detailed explanation, the corresponding C code is accompanied.

ADT *Set* is

Objects: a *Setpointer* head and a structure *Set* composed of 1). *Element* e 2) *Setpointer* link, pointed by a head.

Functions:

For all *s1*, *s2*, *u1*, *is1*, *diff1* $\in$ *Setpointer*, *e* $\in$ *Element*, *TRUE*, *FALSE* $\in$ *Boolean*

And the operations *, ->, != and == are same with operations in general C code.

Result Type	Function	Definition

Setpointer	Create()	::= Set a Setpointer head as NULL and return head. Currently its head has no address, which refers to an empty set.
		`Set * head = NULL;`
		`return head;`

Boolean	IsEmpty(&*s1*)	::= check whether *s1*'s head is an empty set or not.
		`if (*head == NULL) return true;`
		`else return false;`

Boolean	IsIn(&*s1*, *e*)	::= check whether *e* is in *x* and if exists, return True.
		`if (*head == NULL) return false;`
		`if ((*head)->e == e) return true;`
		`return IsIn(&(*head)->link, e);`

Boolean	Insert(&*s1*, *e*)	::= if there's no duplicate element, then insert *e* in *s1*. if there is one or more elements in this set then its

		head becomes its link repeatedly. So, this function is recursively implemented.
		`if (IsEmpty(&(*head)))`
		`{`
		`*head = (Set *)malloc(sizeof(Set));`
		`(*head)->e = e;`
		`(*head)->link = NULL;`
		`return true;`
		`}`
		
		`if ((*head)->e == e)`
		`return false;`
		
		`return Insert(&(*head)->link, e);`

Boolean	Remove(&*s1*, *e*)	::= if there's element *e* in *s1*, remove *e* in *s1* and connect. It is also recursively implemented as follows.
		`if (*head == NULL) return false ;`
		`if ((*head)->e == e)`
		`{`
		`Set * tmp = (*head)->link;`
		`free(*head);`
		`*head = tmp;`
		`return true;`
		`}`
		`return Remove(&(*head)->link, e);`

SetPointer	Union(&*s1*, &*s2*)	::= Create a Setpointer u1 and insert all of elements in set x and y, using function Insert. If there is a duplicate element, then it is skipped while Insert is implemented. And then return this Union u1.
		`Set * u1 = Create();`
		`Set * tmp1 = *s1;`
		`Set * tmp2 = *s2;`
		`for (tmp1->e; tmp1 != NULL; tmp1 = tmp1->link)`
		`Insert(&(u1), tmp1->e);`
		`for (tmp2->e; tmp2 != NULL; tmp2 = tmp2->link)`
		`Insert(&(u1), tmp2->e);`
		`return u1;`

SetPointer	Intersection(&*s1*, &*s2*)	::= Create a Setpointer is1 and if s1 and s2 have same elements, then insert them into Intersection by implementing function IsIn and Insert. And then return this Intersection is1.
		`Set * is1 = Create();`
		`Set * tmp1 = *s1;`
		`for (tmp1->e; tmp1 != NULL; tmp1 = tmp1->link){`
		`if (IsIn(&(*s2), tmp1->e) == true)`
		`Insert(&(is1), tmp1->e);}`
		`return is1;`

SetPointer	Difference(&*s1*,	::= Make a Setpointer diff1, insert all of elements in s1

	&s2)	into diff and then remove all of elements in y from diff. if there is no duplicate element in diff, then it is skipped while Remove is implemented.
		`Set * diff1 = Create();`
		`Set * tmp1 = *s1;`
		`Set * tmp2 = *s2;`
		`for (tmp1->e; tmp1 != NULL; tmp1 = tmp1->link)`
		`Insert(&(diff1), tmp1->e);`
		`for (tmp2->e; tmp2 != NULL; tmp2 = tmp2->link)`
		`Remove(&(diff1), tmp2->e);`
		`return diff;`

Bag: A collection of 1). finite elements, which could be number, name, people and so on, 2). with no particular order. It could contain duplicate elements unlike set. Especially, a set with no elements is defined as an empty set.

In this ADT, Bag is implemented by dynamically linked list so there is no in insertion failure due to lack of storage places. Even if there is duplicate element in set in advance, new element must be surely inserted: likewise, if there are duplicate elements in set, then they are all removed from set in only one Remove invocation. For detailed explanation, the corresponding C code is accompanied.

ADT *Bag* is

Objects: a *BagPointer* head and a structure *Bag* composed of 1). *Element* e 2) *BagPointer* link, pointed by a head.

Functions:

For all $b \in$ *BagPointer*, $e \in$ *Element*, *TRUE*, *FALSE* $\in$ *Boolean*

And the operations *, ->, != and == are same with operations in general C code.

Result Type	Function	Definition

SetPointer	Create()	::= Set a Bagpointer head as NULL and return head. Currently its head has no address, which refers to an empty set.
		`Bag * head = NULL;`
		`return head;`

Boolean	IsEmpty(&*b*)	::= check whether *b*'s head is an empty set or not.
		`if (*head == NULL) return true;`
		`else return false;`

Boolean	IsIn(&*b*, *e*)	::= check whether *e* is in b and if exists, return True.
		`if (*head == NULL) return false;`
		`if ((*head)->e == e) return true;`

		`return IsIn(&(*head)->link, e);`

Void	Insert(&*b*, *e*)	::= even if there's duplicate element, *e* must be inserted *e* in *b*. this function is recursively implemented and you can understand by replacing the name head below with *b*.
		`if (IsEmpty(&(*head)))`
		`{`
		`*head = (Set *)malloc(sizeof(Set));`
		`(*head)->e = e;`
		`(*head)->link = NULL;`
		`return;`
		`}`
		`Insert(&(*head)->link, e);`

Boolean	Remove(&*b*, *e*)	::= Firstly check whether *e* is in *b* or not with IsIn. If not, this function ends at that point and returns Boolean variable false. If there's one or more *e* in *b*, you can check its element from head to its link. Implemented recursively, head becomes its link and can insist what element is in value pointed by current *b*'s head and if you find two elements same, remove it from *b* and connect: at that moment you have to check whether *e* still remains in bag *b* by implementing IsIn once more. If not, then this function finally ends, returning Boolean variable true.
		`if (IsIn(&(*head), e) == 0) return false;`
		`else if ((*head)->e == e)`
		`{`
		`Set * tmp = (*head)->link;`
		`free(*head);`
		`*head = tmp;`
		`if (IsIn(&(*head), e) == 1) return`
		`Remove(&(*head), e);`
		`else return true;`
		`}`
		`return Remove(&(*head)->link, e);`

Problem #1-2

Following is a Step Count Table of Problem #1-2 Function.

MAX_SIZE is replaced by n, since its length is too long.

Statement	s/e	Frequency	Total Steps
`void multi(int a[][n], int b[][n], int c[][n])`	0	0	0
`{`	0	0	0
`  int i, j, k;`	0	0	0
`  for (i = 0; i < n; i++) {`	1	n+1	n+1
`    for (j = 0; j < n; j++) {`	1	n(n+1)	n^2+n
`      c[i][j] = 0;`	1	n^2	n^2
`      for (k = 0; k < n; k++) {`	1	$n^2(n+1)$	n^3+n^2

c[i][j] += a[i][k] * b[k][j];
}
}
}
}
	1
	0
	0
	0
	0
		n^3
		0
		0
		0
		0
			n^3
			0
			0
			0
			0

Total			**$2n^3+3n^2+2n+1$**

Problem #1-3

Show that the following statements are incorrect:

(2^{nd} and 3^{rd} problems are identical in #1-3 so following are just 5)

1. $10n^2 + 9 = O(n)$

: Let the left expression be *f(n)*. Iff there exist positive constants *c* and n_0 such that $f(n) \leq cn$ for all *n*, $n \geq n_0$, then this statement is correct.

Whenever n is greater than *c/10*, *f(n)* becomes greater than *cn*. Therefore, there are no positive constants *c* and n_0 such that satisfy definition of Big O.

Correct statement: $10n^2 + 9 = O(n^2)$

2. $n^2 \log n = \theta(n^2)$

: Let the left expression be *f(n)*. Iff there exist positive constants c_1, c_2 and n_0 such that $c_1n^2 \leq f(n) \leq c_2n^2$ for all *n*, $n \geq n_0 > 0$, then this statement is correct.

Whenever n is greater than or equal to 2, *f(n)* cannot be less than or equal to c_2n^2 in any c_1. So this statement is incorrect.

Correct statement: $n^2 \log n = \theta(n^2 \log n)$

3. $n^2 / \log n = \theta(n^2)$

: Let the left expression be *f(n)*. Iff there exist positive constants c_1, c_2 and n_0 such that $c_1n^2 \leq f(n) \leq cn^2$ for all *n*, $n \geq n_0 > 0$, then this statement is correct.

Whenever *n* is greater than $10^{1/c1}$ *f(n)* cannot be greater than or equal to c_1n^2 for *n* greater than or equal to n_0. So this statement is incorrect.

Correct statement: $n^2 / \log n = \theta(n^2 / \log n)$

4. $n^3 2^n + 6n^2 3^n = O(n^2 2^n)$.

: Let the left expression be *f(n)* and the right expression in Big O be *g(n)*. Iff there exist positive constants *c* and n_0 such that $f(n) \leq cg(n)$ for all *n*, $n \geq n_0$, then this

statement is correct.

According to Big O definition of this statement, there must exist some examples like $f(n) \leq cg(n)$ for $n \geq n0$, but $n^3\ 2^n + 6n^2\ 3^n > n^2\ 2^n$ for $n \geq 1$. In fact, there is so large gap between 3^n and 2^n. So this statement is incorrect.

Correct statement: $n^3\ 2^n + 6n^2\ 3^n = O(3^n)$

5. $3^n = O(2^n)$.

: Let the left expression be $f(n)$. Iff there exist positive constants c and n_0 such that $f(n) \leq c2^n$ for all n, $n \geq n_0$, then this statement is correct.

According to Big O definition of this statement, there must be some examples like $f(n) \leq c2^n$ for $n \geq n_0$, but there is so large gap between 3^n and 2^n. No matter how we apply large c in this statement, this constant c cannot exceed exponential increase. For instance, $3^n < 10000 \times 2^n$ is only possible when $n < 23$.

Even if we push very big c which satisfies this expression, it cannot be a useful notation of Big O. So this statement is incorrect.

Correct statement: $3^n = O(3^n)$.