Анализ на Задача 5, 2005/2006

Въведение

Задача 5 на пръв поглед изглежда тривиална, но е от изключителна важност каква структура ще се използва за съхранение на данните и с каква сложност са операциите върху структурата. С времевото ограничение от 2 сек. на тест, решения със сложност по-висока от O(sqrt(N)) за която и да е операция не биха постигнали максималният резултат от 100 точки.
За решение на задачата авторът е избрал модификация на Случайно Балансирано Двоично Дърво за Търсене (Randomized Binary Search Tree, RBST). RBST са описани от Conrado Martinez и Salvador Roura през 1997 в [1].

С помощта на модифицирано RBST се постигат следните сложности за различните операции:

Операция	Сложност
split	$O(\log_2 N)$
merge	$O(\log_2 N)$
min	O(1)
max	O(1)
reverse	O(1)
create	O(1)

Описание на структурата

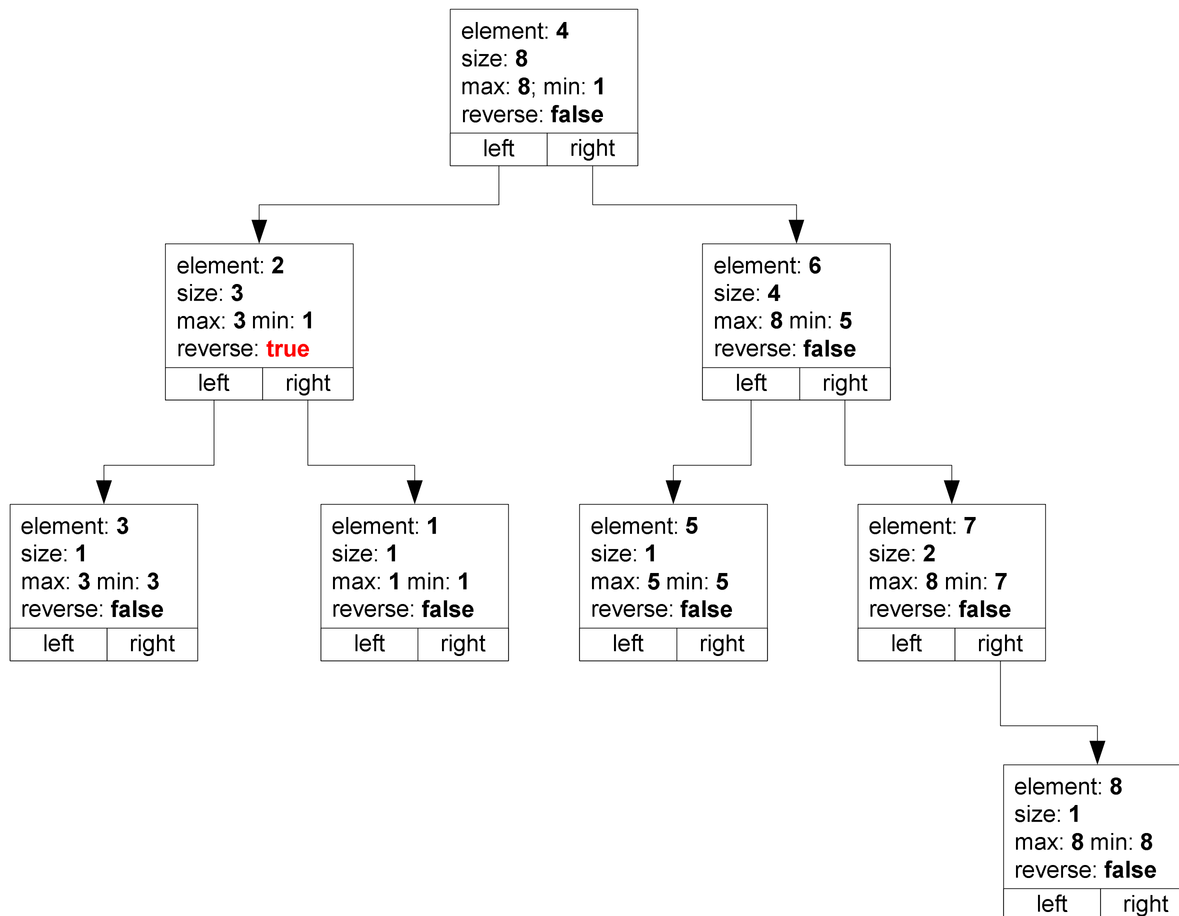
Използва се модифицирано RBST като за текущ връх със стойност X, в лявото под дърво се намират всички елементи преди X (при последователна наредба), а в дясното – всички елементи след X. Един връх в дървото се описа с помощта на следната структура:

```
struct Node
{
	int size;
	int element;
	Node* left;
	Node* right;
	int min;
	int max;
	bool reverse;
}
```

където:

- size – броят на всички елементи (рекурсивно, включително и текущия) под текущия елемент
- element – стойността на текущия връх.
- left, right – лявото и дясното под дърво.
- min, max – минималният и максималният елемент (рекурсивно, включително и текущия) под текущия елемент.
- reverse – редът на елементите под текущия.
    - При reverse == true – елементите са в обратно-последователна наредба.

Пример:
Следната структура описва редицата 1,2,3,4,5,6,7,8.



Описание на операциите

Ще разгледаме операциите create, min, max, reverse, split и merge.
Всеки един списък ще е корен на RBST – т.е Node. Списъците са номерирани с числата от 1 до 1000000 следователно могат да се индексират директно от масив с 1000000 елемента. В реализацията:

```
static Node* lists[MAX_NUMBER_OF_LISTS];
```

Следователно намирането на списък (корен на RBST) по номер ще отнема O(1) време.

create

Създаването на нов списък е свързано със създаването на RBST със само един елемент. Ако предположим че създаваме списък с елемент X, тогава:

left = right = NULL
min = max = element = X
reverse = false
size = 1

Пример:
Операцията:

```
create 1 8
```

ще създаде следното RBST:

element: **8** size: **1** max: **8** min: **8** reverse: **false**	
left	right

min, max

Операциите min и max връщат минималният и максималният елемент от корена на дървото – т.е член-променливите min и max от Node.

reverse

При операция reverse, член-променливата на корена на дървото reverse приема стойността на логическото си отрицание. При създаване на списък, reverse има стойност false.

Пример:

`create 1 8`	`reverse 1`	`reverse 1`
element: **8** size: **1** max: **8** min: **8** reverse: **false** left | right	element: **8** size: **1** max: **8** min: **8** reverse: **true** left | right	element: **8** size: **1** max: **8** min: **8** reverse: **false** left | right

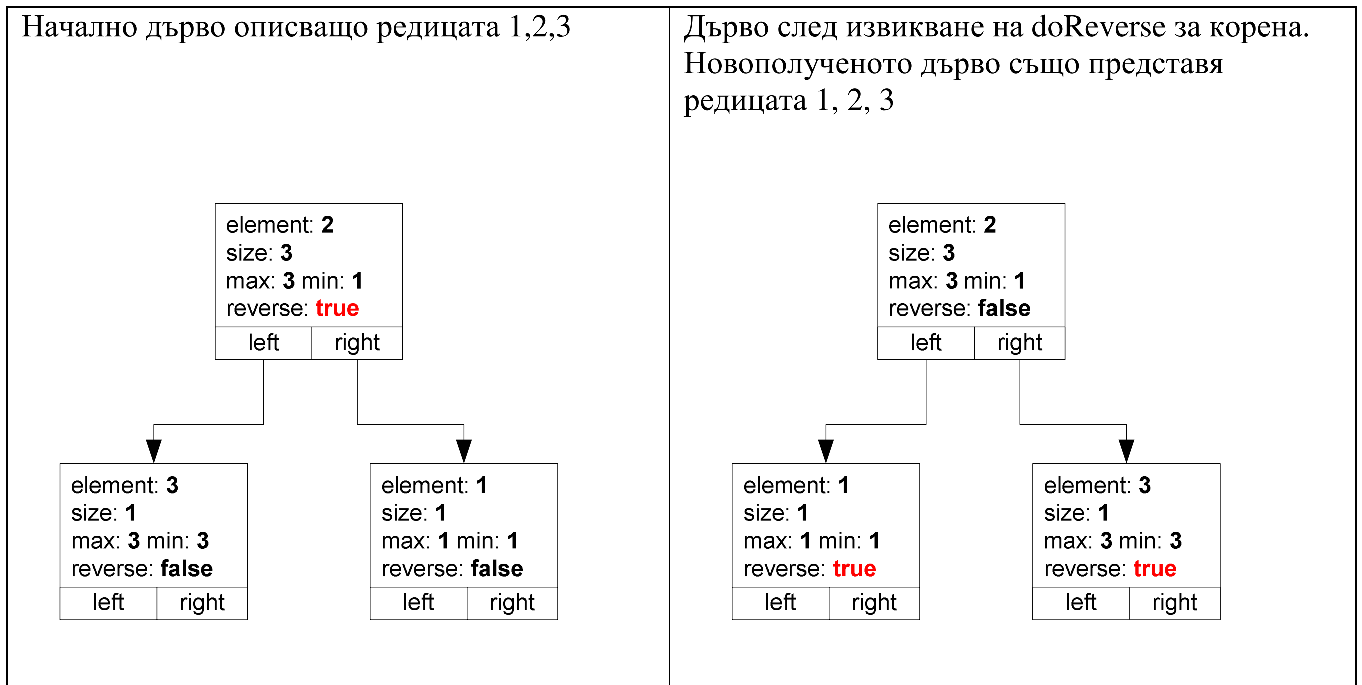
Стойности на reverse за всеки връх от дървото трябва да се взима в предвид при операциите merge и split. За да не се разглеждат по няколко случая при тези операции (merge и split) който биха довели до усложнения и възможности за грешки, при достигане на връх от дървото, при който reverse = true, сменяме лявото с дясното под дърво, и обръщаме (правим операции reverse) лявото и дясното под дърво. Методът:

```
void doReverse()
{
	if (reverse)
	{
		Node* tmp = left;
		left = right;
		right = tmp;
		reverse = false;

		if (left != NULL) left->reverse = !left->reverse;
		if (right != NULL) right->reverse = !right->reverse;
	}
}
```

на Node се грижи за това обръщане. Този метод гарантира, че дървото ще си запази структура и реда и че стойности на reverse ще е false след изпълнението му.

Пример:



При описването на операциите split и merge ще разгледаме само случай, за които reverse = false. Ако reverse = true за който и да е връх, винаги можем да извикаме doReverse () при което reverse става false, но дървото запазва правилната последователност на елементите.

split

Операцията е подобна на описаната в [1] и [2] като е модифицирана за целите на задачата.

Нека K е позицията, от която правим разделянето на списъка на 2 части (от 1 до K-1 и от K до N). Нека left и right са съответно резултатните дървета 1..K-1 и K..N, а root е корена на текущото дърво.
Тогава функцията ще изглежда така:

```
void split(Node **left, Node **right, Node* root,....)
```

Алгоритъмът е рекурсивен и е следният:

1. Ако K е в дясното под дърво, тогава left = root и извикваме рекурсивно:
   ```
   split(&((*left)->right), right, root->right...);
   ```
   Т.е цялото ляво под дърво е за left; left->right и right рекурсивно си поделят дясното под дърво.
2. Ако K е в лявото под дърво, тогава right = root и извикваме рекурсивно:
   ```
   split(left, &((*right)->left), root->left...);
   ```
   Т.е цялото дясно под дърво е за right; right->left и left рекурсивно си поделят лявото под дърво.

3. Ако K съвпада с позицията на корена, тогава:

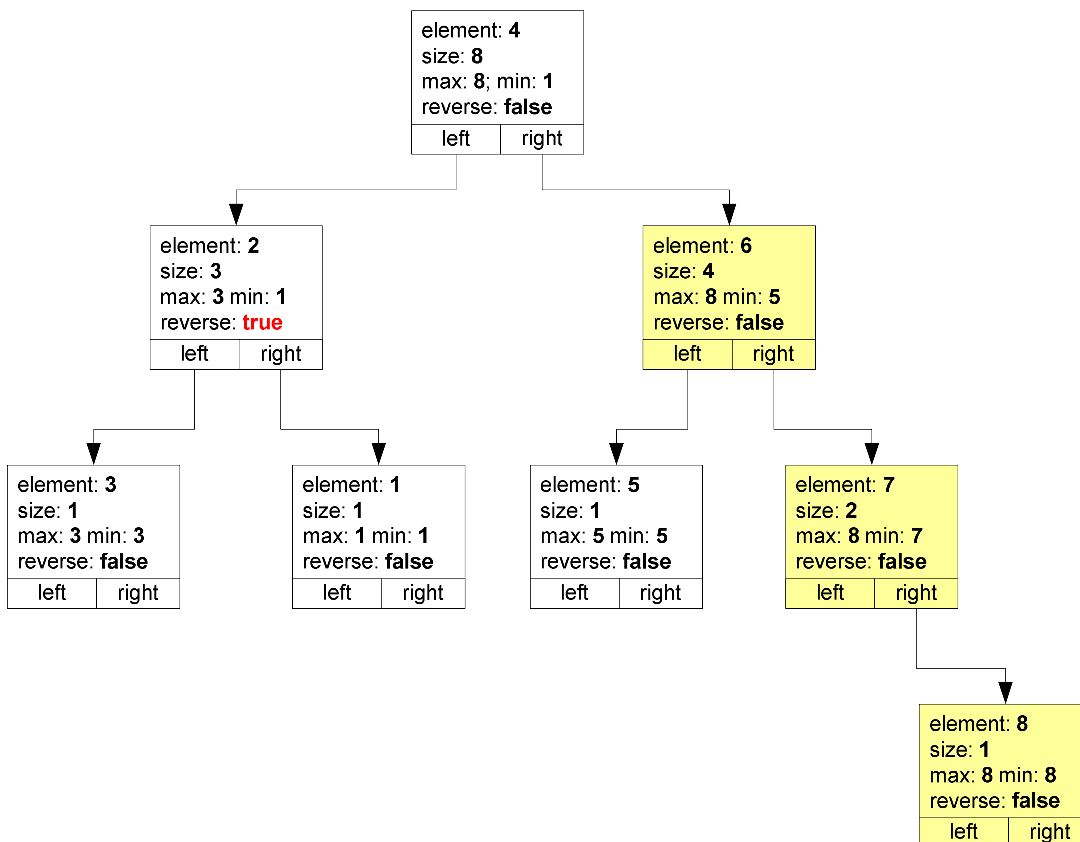
```
*left = root;
*right = root->right;
root->right=NULL;
```

Т.е – left взима корена, а right взима дясното под дърво.

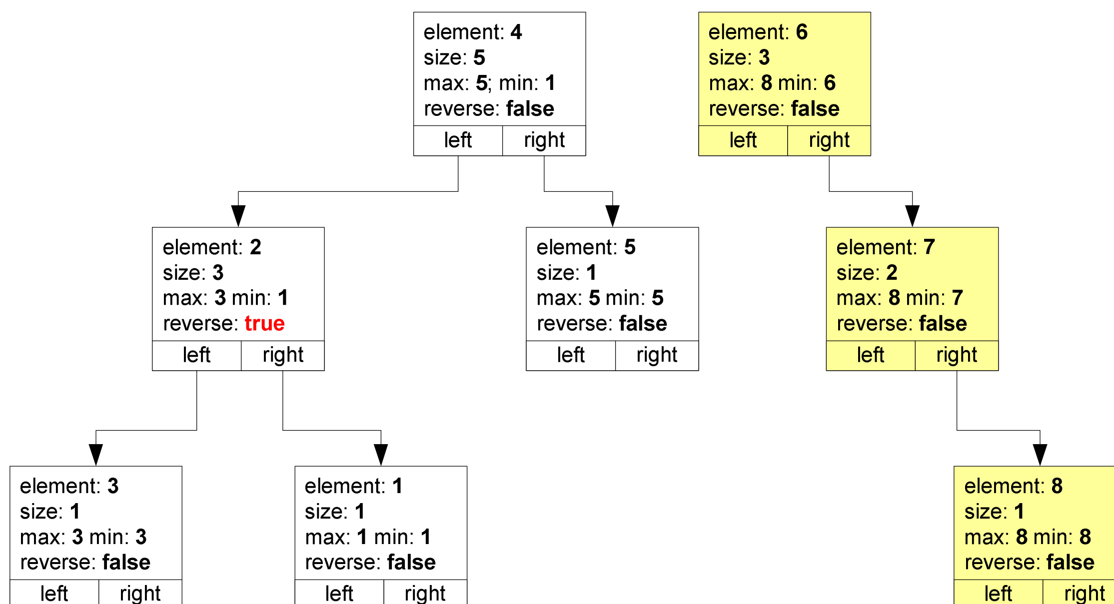
При излизане от съответното ниво на рекурсията се грижим за правилната промяна на min, max и size. Извиква се:

```
void updateSizeAndMinMax(Node *root)
```

Нека K = 6 т.е split трябва да раздели дървото по-долу на 2 части. Едната част ще съдържа редицата 1,2,3,4,5 а другата 6,7,8.



След изпълнението на split дървото ще се раздели на 2 части:



merge

Операцията е подобна на описаната в [1] и [2] като е модифицирана за целите на задачата. Нека left и right са 2 дървета, които трябва да се обединят. Тогава функцията за обединяване ще изглежда така:

```
Node* merge(Node* left, Node* right)
```

Резултатът е обединеното дърво.

Алгоритъмът е рекурсивен и е следният:

1. С определена вероятност избираме 2 или 3
2. Избираме left за корен на резултатното дърво и
   ```
   left->right = merge(left->right, right);
   ```
   Т.е рекурсивно обединяваме left->right и right.
3. Избираме right за корен на резултатното дърво и
   ```
   right->left = merge(left, right->left);
   ```

Оптимизации

За по-добро бързодействие са реализирани следните 2 оптимизации:

- Използване на статична памет – т.е паметта за всички Node, които са не повече от 200000 се заделя от операционната система. Така се избягва заделяне на heap памет.
- Оптимизиран вход/изход. Операциите fscanf и fprintf са бавни, затова не се използват. Целият входен файл се прочита в голям буфер и се обработва символ по символ. Също то е валидно и за изходният файл.

Използвана литература

[1] C.Martinez, S.Roura. *Randomized Binary Search Trees*.
(http://www.lsi.upc.es/~conrado/research/papers/jacm-mr98.pdf)

[2] G. Blelloch, M. Reid-Miller. *Fast Set Operations Using Treaps*
*(*http://citeseer.ist.psu.edu/blelloch98fast.html)