University of Ljubljana
Institute of Mathematics, Physics and Mechanics
Department of Mathematics
Jadranska 19, 1111 Ljubljana, Slovenia

Preprint series, Vol. 41 (2003), 889

A STATIC DATA STRUCTURE FOR DISCRETE ADVANCE BANDWIDTH RESERVATIONS ON THE INTERNET

Andrej Brodnik Andreas Nilsson

ISSN 1318-4865

August 28, 2003

Ljubljana, August 28, 2003

A Static Data Structure for Discrete Advance Bandwidth Reservations on the Internet

Andrej Brodnik [*] [†] Andreas Nilsson [†]

Abstract

In this paper we present a discrete data structure for reservations of limited resources. A reservation is defined as a tuple consisting of the time interval of when the resource should be reserved, I_R, and the amount of the resource that is reserved, B_R, formally $R = \{I_R, B_R\}$.

The data structure is similar to a segment tree. The maximum spanning interval of the data structure is fixed and defined in advance. The granularity and thereby the size of the intervals of the leaves is also defined in advance. The data structure is built only once. Neither nodes nor leaves are ever inserted, deleted or moved. Hence, the running time of the operations does not depend on the number of reservations previously made. The running time does not depend on the size of the interval of the reservation either. Let n be the number of leaves in the data structure. In the worst case, the number of touched (i.e. traversed) nodes is in any operation $O(\log n)$, hence the running time of any operation is also $O(\log n)$

1 Introduction

The original, never published version of this paper was called *"An Efficient Data Structure for Advance Bandwidth Reservations on the Internet"*. The original paper was referred to in the paper "Performance of QoS Agents for Provisioning Network Resources" ([10]) by Schelén et. al. under the reference number 14, but the reference should really be changed to the current paper.

2 Definition of the problem

The problem we deal with, we call *"The Bandwidth Reservation Problem"*. A reservation is a time interval during which we reserve constant bandwidth. The solution given here works in a discrete bounded universe.

By a discrete bounded universe we mean a universe with a limited duration and a fixed time granularity. By using a fixed granularity we have divided the time into time slots (frames). We use a slotted time in hope to get a smaller data structure, faster operations and hence gain benefits of a larger aggregation. This hope is inspired by the fact that problems are generally easier to solve in a bounded (discrete) universe than in the general case ([4]).

[*] Department of Theoretical Computer Science, Institute of Mathematics, Physics, and Mechanics, Ljubljana, Slovenia

[†] Department of Computer Science and Electrical Engineering, Luleå University of Technology, Luleå, Sweden

We observe that the bandwidth reservation problem is constrained by the physical world and therefore no reservations will occur in the past and very few in a distant future.

Throughout the paper we use the following notation:

- We have a bounded maximum interval M starting at S_M and ending at E_M, hence $M = [S_M, E_M]$. M is divided into fixed size time slots of size g. The size of the interval M is denoted by $|M|$.

- In general, a discrete interval, I, is defined as the duration between a starting point S and an ending point E, and the interval is divided into discrete slots of size g. In short, $I = [S, E]$. Moreover, since $S_M \leq S < E \leq E_M$ $I \subseteq M$.

- The bandwidth is denoted by B. A reservation, R, is defined by an interval I and a (constant) amount of reserved bandwidth B, during I. In short, reservation is a tuple $R = \{B, I\}$. Items related to reservations are denoted by a subscript, e.g. $R = \{B_R, I_R\}$.

- A data structure storing reservations made is denoted by $\mathcal{D}$. An item related to a "query" toward the data structure is denoted by a subscript Q, e.g. $I_Q = [S_Q, E_Q]$.

The bandwidth reservation problem defines three operations. First, we have a query operation that only makes queries of the kind: "How much bandwidth is reserved at most between time S and time E?". Further, we have update operations: an insertion of a new reservation and a deletion of a reservation already made. Formally these operations define:

Definition 1 *We have a bounded maximum interval M divided into time slots of size g. Let a reservation, $R = \{B_R, I_R\}$, be on an interval $I_R \subseteq M$ with an associated bandwidth, B_R. Then the* bandwidth reservation problem *requires the following operations:*

`Insert`$(\mathcal{D}, R)$*, which increases the reserved bandwidth during the interval I_R for B_R.*

`Delete`$(\mathcal{D}, R)$*, which decreases the reserved bandwidth during the interval I_R for B_R.*

`MaxReserved`$(\mathcal{D}, I_Q)$ *which returns the maximum reserved bandwidth, during the interval I_Q.*

Note, deletion is the same as an insertion but with a negative bandwidth.

2.1 Background of the problem

The bandwidth reservation problem is not so well studied in the literature. On the other hand, two related problems, the partial sum problem ([5], brief in [8]), and the prefix sum problem ([5]), are. In the partial sum problem we have an array $V(i), 1 \leq i \leq n$ and want to perform these two operations: (1) `update`: $V(i) = V(i) + x$; and (2) `retrieve`: $\sum_{k=1}^{m} V(k)$ for arbitrary values of i, x and m. There is only a slight difference between the partial sum problem and the prefix sum problem, in the prefix sum problem the query always starts at the beginning and in the partial sum problem the queries are for an arbitrary interval.

In our solution we will use a data structure similar to segment trees ([9]). Segment trees represent a method for storing set of intervals. For instance, we have m intervals with n unique starting or ending points. The segment tree is then an efficient data structure for storing those intervals and answering queries over which of the m intervals spans the query interval. Formally, let $\mathcal{S}$ denote a set of m intervals with n unique starting and ending points. Let Z_i be a starting or ending point for an interval in $\mathcal{S}$. Let U be our universe where $U = \{Z_1, Z_2, ..., Z_n\}$ given that $Z_{i-1} < Z_i$ where $1 < i \leq n$. The leaves in the segment tree correspond to the intervals: $(-\infty, Z_1), [Z_1, Z_1]; (Z_1, Z_2), [Z_2, Z_2]; ...;$
$(Z_n, +\infty)$ as shown in Fig. 1.

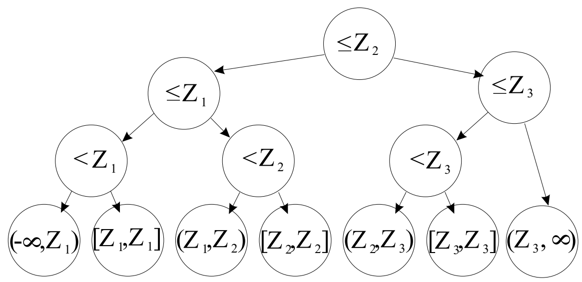

Figure 1: An example of a segment tree.

An internal node represents the interval of the tree rooted at it and it also stores the information about the interval of its left subtree. The interval of the node is the union of the intervals of its children. Each node (and a leaf) contains a list of pointers to previously inserted intervals that completely cover the interval of the node but not the interval of the node's parent. During the insertion of the interval I, the pointer to the interval I is inserted in a node N's list, if all children of N have their corresponding intervals within I and the parent of N does not. Consequently, pointers to an interval are stored at maximum two nodes on each level of the tree. The segment tree has $2n+1$ leaves and $2n$ nodes. Since the nodes in the tree have constant number of children the height of the tree is $O(\log n)$. This is also the complexity of an insertion and a query.

3 Solution

Our solution is a modified segment tree ([9]). In our data structure, Advanced Segment Tree (AST), each node represents one time interval. Every node in the tree consists of the interval it represents, pointers to each of the node's children, and two values (described more thoroughly further down).

The interval to which the root corresponds is M. Let L denote the number of levels that the data structure consists of. All nodes on level l have time intervals of the same size and they do not intersect. They follow consecutively one another. This means that the union of all intervals on level l is M. Each level has a divisor that tells the number of children that a node on that particular level has. The divisors are gathered up from the root to the leaves in a set $X = \{X_1, X_2, ..., X_{L-1}\}$, where X_1 is the number of the root's children. The divisor X_l does not only tell the number of children that a node has, but also the size of the interval $|M_l|$ on level l:

$$|M_l| = \begin{cases} |M| & i = 1 \\ \frac{|M_{l-1}|}{X_l} & 1 < l \leq L \end{cases} \tag{1}$$

Consequently the number of nodes on level l is

$$n_l = \begin{cases} 1 & l = 1 \\ \prod_{i=1}^{l-1} X_i & 1 < l \leq L \end{cases} . \tag{2}$$

and the number of leaves of the complete data structure is

$$n = n_L = \prod_{i=1}^{L-1} X_i \ . \tag{3}$$

The divisors X_i must be set so that $g = \frac{|M|}{n}$, where n is defined in eq. (3) and where g is the time granularity (size of the leaves). Hence, the choices of $|M|$, g, L, n and X are related. For instance, choosing $|M|$ to be a prime number makes the tree consist of only two levels, the top and the leaf level. We get the simplest tree when $|M| = 2^L \cdot g$, i.e. $X = \{X_l \mid X_l = 2, \text{for } 0 < l < L\}$. Note that the fundamentals of the data structure do not change if the values of the divisors change. There will however be a deterioration in performance for each added level. The tree is only built once and therefore the tree is always perfectly balanced. There is a difference between the segment tree and our data structure regarding the leaves. In the segment tree there is a leaf for the open interval, (X_{i-1}, X_i), as well as the closed interval, $[X_i, X_i]$, but in our data structure leaves represent semi-open intervals $(X_i, X_{i+1}]$. To describe our data structure we use the following notation:

- Let N denote "the current node" during a description of a traversal of the tree. Let N_L denote the leftmost child of N, and N_R the rightmost child.
- Each node, N, stores the interval $I_N = [S_N, E_N]$ that the node subtends.
- Each node, N, stores the amount of bandwidth, nv_N, that was reserved over exactly the whole interval I_N.
- Each node N also stores the maximum value of reserved bandwidth excluding the value nv_N on the interval I_N. This maximum value is denoted as mv_N.

```
#define Split {2,2,2,2,2,3,2,2,2,3,2,2}
#define MaxSplit 3
typedef struct NODE {
   interval_type Interval;
   int           node_value;
   int           max_value;
   struct NODE *Children[MaxSplit];
} AST_type;
```

Algorithm 1: Advance Tree Definitions in C

In Fig. 2 is given an example of how to make the data structure subtending a 32-day month, with g representing 5 minutes. All nodes have 2 or 3 children in order to have the wanted interval sizes. Hence, in Fig. 2 $X = \{2, 2, 2, 2, 2, 3, 2, 2, 2, 3, 2, 2\}$.

Fig. 3 presents an example of a bandwidth utility graph. The graph shows the amount of bandwidth that is reserved over time. In Fig. 4 we are showing the values in the tree corresponding to the example graph from Fig. 3. Fig. 4, also shows how mv is calculated.

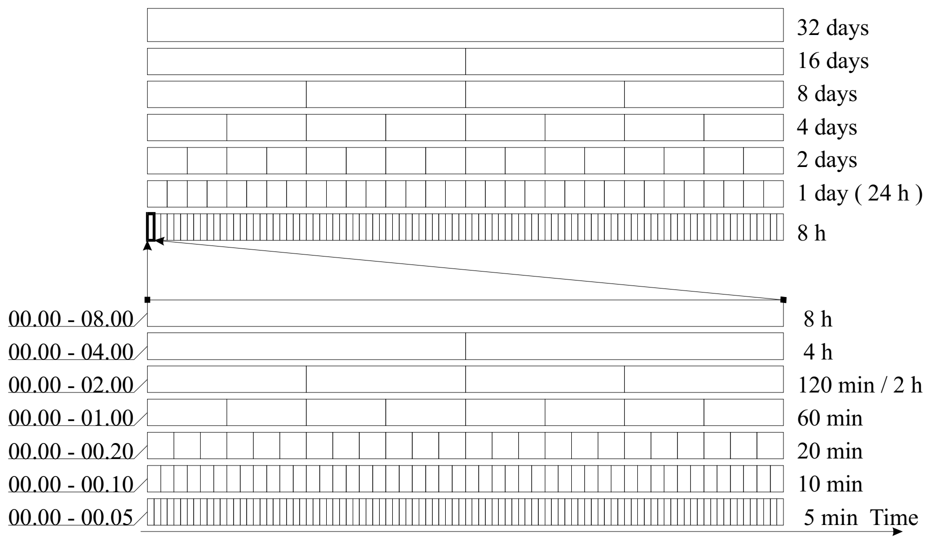

Figure 2: An example of the data structure.

To describe the operations from Definition 1 we use the data structure formally defined in Algorithm 1. We start by describing the function $\texttt{MaxReserved}(N, I_Q)$.

- If the interval of node N satisfies $I_Q = I_N$ (i.e. $S_N = S_Q$ and $E_N = E_Q$) then $nv_N + mv_N$ is returned as the result.
- If I_Q is entirely contained within the interval of child N_C of N, then the returned value will be: $nv_N + \texttt{MaxReserved}(N_C, I_Q)$.
- If I_Q spans the intervals of more than one child of N, I_Q is divided into one part for each of the m children of N which intervals I_Q at least partially spans - i.e. $I_{Q_i} = I_Q \cap I_{N_{Ci}}$, for $1 \leq i \leq m$ where I_{Q_1} is the leftmost interval and $I_{N_{C1}}$ is the interval of the leftmost child of N that has an interval that at least partially spans I_Q. The Fig. 5 illustrates the split of I_Q into smaller intervals.

  The returned value will be

$$\max_{1 \leq i \leq m} (\texttt{MaxReserved}(N_{Ci}, I_{Qi}))$$

  and it can be more efficiently computed as:

$$\max\left(\begin{array}{l} \texttt{MaxReserved}(N_{C1}, I_{Q1}) \quad , \\ \max_{1<i<m} (nv_{N_{Ci}} + mv_{N_{Ci}}) \quad , \\ \texttt{MaxReserved}(N_{Cm}, I_{Qm}) \end{array} \right) \tag{4}$$

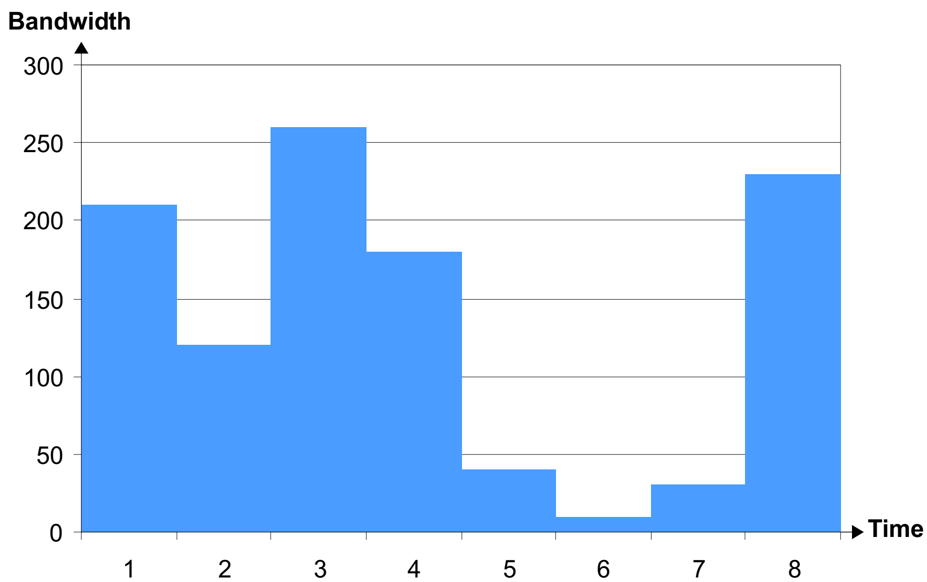

Figure 3: Bandwidth - Time graph.

In Fig. 6 an insertion of a reservation of B_R units of bandwidth is shown. The figure also shows which nodes will be touched when a `MaxReserved` function is invoked with the same query interval. The `MaxReserved` function is formally defined in Algorithm 2.

The function $\texttt{Insert}(N, \{I_R, B_R\})$ works in a similar way as the `MaxReserved` function. `Insert` must also verify that the inserted reservation does not result in an over-reservation of bandwidth. This verification can be done by making the reservation, then performing a $\texttt{MaxReserved}(N, I_R)$ query, and finally comparing with the maximum reservable bandwidth on the link. If an over-reservation occurs the reservation must be removed. More efficient is to let the `Insert` function perform the check during its execution. We will describe the recursive `Insert` function without integration of `MaxReserved` functionality, which inclusion is trivial and is shown in Section 3.2.2.

- If the interval of node N satisfies that $I_R = I_N$ (i.e. $S_N = S_R$ and $E_N = E_R$), then nv_N is increased by B_R.
- If I_R is entirely contained within the interval of one child N_C of N, then the function $\texttt{Insert}(N_C, \{I_R, B_R\})$ is called and when it returns the mv_N is updated according to the equation

$$mv_N = \max_{1 \leq i \leq k} (nv_{Ci} + mv_{Ci}) \quad , \tag{5}$$

  where k is the number of children that N has.
- If I_R spans the intervals of more than one child of N, I_R is divided exactly as the query interval into one part for each of the m children of N which intervals I_R at least partially spans - i.e. $I_{R_i} = I_R \cap I_{N_{Ci}}$, for $1 \leq i \leq m$. The `Insert` function is called once for each of the m children N_{Ci}, $\texttt{Insert}(N_{Ci}, \{I_{R_i}, B_R\}), 1 \leq i \leq m$. When the calls return, the mv_N is updated as shown in eq. (5).

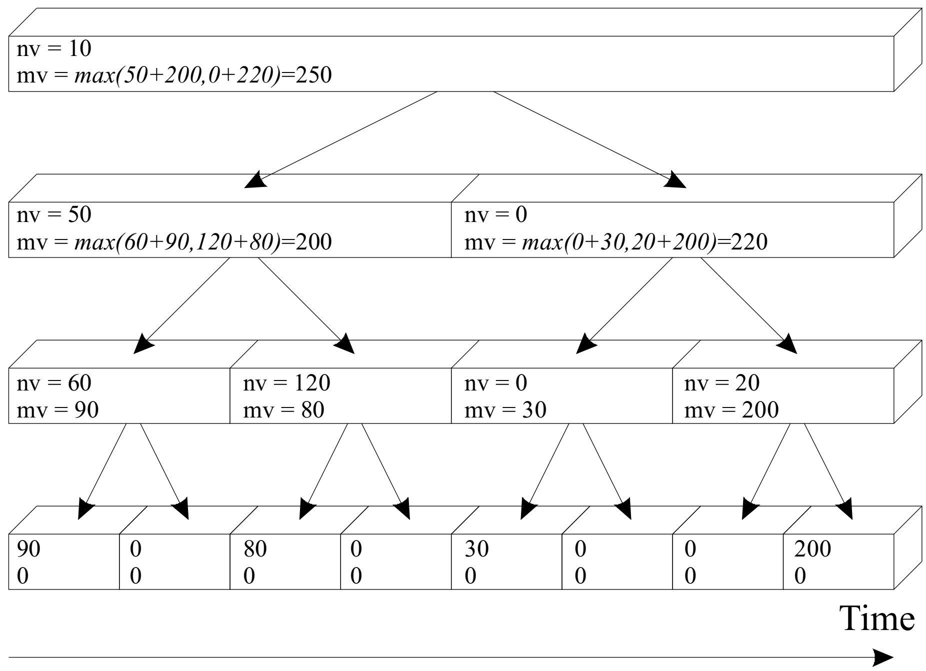

Figure 4: An example showing values in the data structure.

In Fig. 6 an insertion of a reservation and the calculation of the new nv's and mv's are shown. The `Insert` function is formally defined in Algorithm 3.

The $\texttt{Delete}(N, \{I_R, B_R\})$ is implemented as a call of `Insert` with B_R negated, Algorithm 4.

The `Insert` function as well as the `MaxReserved` function only traverses the tree twice from the top to the bottom. Once for the rightmost part of the interval I_Q and once for the leftmost part. For the middle part of the interval the recursion never goes deeper than 1 level. The update of the mv values is done during the traversal so no further work is needed. For a tree where every node only has two children, both functions will touch at the most $4 \cdot (\lg n) - 7$ nodes. For a tree with other divisors the constants are different but still the running time remains $O(\log n)$. Even if the check for over-reservations is implemented as a separate call to the `MaxReserved` function the running time remains within $O(\log n)$.

Theorem 1 *The running time for all operations solving the bandwidth reservation problem as defined in Definition 1 and using AST, is* $O(\log n)$.

3.1 Implicit data structure

The obvious way to implement our data structure is to use pointers between the nodes. Since our data structure is built only once and the nodes never change, it is possible to store the data structure in an implicit way in an array. In an implicit data structure the positions of the nodes are implicitly defined. We use the set X, which tells the number of children each node has on level l (see eq. (1)), to compute the positions of the nodes. Once the array is built we can use X to calculate the index of the node instead of using

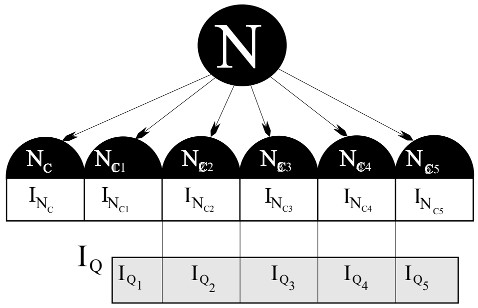

Figure 5: An example showing a split of an interval (circles represent nodes and rectangles represent intervals).

pointers to traverse the tree. To calculate δ_l, the number of nodes on level l and above, we get from eq. (3):

$$\delta_l = \begin{cases} 1 & l = 1 \\ \sum_{j=2}^{l} \left(\prod_{i=1}^{j-1} X_i \right) + 1 & 1 < l \leq L \end{cases} \tag{6}$$

We index the elements in the array from 1 to δ_L and order the nodes level by level (cf. standard heap order). Consequently, the index of the first element on level l, is the number of nodes in the tree on all previous levels, plus 1, i.e. $\delta_{l-1} + 1$. We store these values in vector σ.

The number of nodes between the node with the index N on level l and the first node on level l is given by $N - \sigma_l$. The number of nodes between the first node on level $l+1$ and N's first child is given by $(N - \sigma_l) \cdot X_k$. The index of the first node on level $l+1$ is given by σ_{l+1}. The number of children that N has is X_l hence N's children indexes γ are:

$$\gamma = \sigma_{l+1} + (N - \sigma_l) \cdot X_l + c, \quad \text{for} \quad 0 \leq c < X_l \tag{7}$$

By using an array instead of using pointers we save the memory for two pointers per node. The execution will be faster due to one memory probe less since vectors X and σ will be in cache.

3.2 Improvements

In this section we describe some performance improvements which should be seen as hints to an implementer of our data structure.

3.2.1 Choice of the intervals

The idea is, to make proper choices about the intervals to improve the running time of the functions. The choices to be made are regarding the duration as well as starting times and ending times. If we are dealing with man made reservations we observe:

- Granularity:
  People will make reservations during times that are logical to them, hence the smallest interval can be 1 minute, 5 minutes, 30 minutes, 1 hour and so on, depending on the application.

```c
bw_type _MaxReserved(AST_type *node,int start, int stop,
        int iLevel) {
   bw_type    MaxBelowNode, MaxBelowNode;
   int         i1;
   AST_type *pRunningNode;
   if (start==node->Interval.Start && stop==node->Interval.End)
      return (node->node_value + node->max_value);
   else {
      MaxBelowNode = 0;
      for (i1=0;i1<Split[iLevel];i1++) {
         pRunningNode = node->Children + i1;
         if ( start < pRunningNode->Inteval.End ) {
            if (pRunningNode->Interval.End < stop) {
               MaxBelowChild = _MaxReserved(pRunningNode, start,
                       pRunningNode->Interval.End, iLevel+1);
               if ( MaxBelowChild > MaxBelowNode )
                  MaxBelowNode = MaxBelowChild;
               start = pRunningNode->Interval.End;
            } else {
               MaxBelowChild = _MaxReserved(pRunningNode,start,
                                             stop,iLevel+1);
               if ( MaxBelowChild > MaxBelowNode )
                  MaxBelowNode = MaxBelowChild;
               break;
            }
         } /* if */
      } /* for */
      return( node->node_value + MaxBelowNode );
   }
}
bw_type MaxReserved(AST_type Data, interval_type I) {
   _MaxReserved(&Data,I.Start,I.End,0);
}
```

Algorithm 2: Advance Tree MaxReserved in C

- Starting points:
  All interval sizes in the tree should be whole minute intervals. For instance, if an interval of 15 minutes should be divided, the divisor 2 is not a proper choice since the children will then have an interval size of 7.5 minutes and those children will in turn be divided to 3.75 minutes, which both will rarely occur. The divisor 3 is in this example a more suitable divisor, which will make the 15 minutes interval divided in 3 intervals of size 5 minutes each.

- Size of intervals:
  If an interval is estimated to be more likely than another, use the more likely choice. For instance, if a 24 hour day is going to be divided, the divisors 2 and 3 seem like good choices. If we estimate that it is more likely that reservations of 8 hours will occur rather than 12 hours, due to the fact that a working day of humans is 8 hours, the divisor 3 should be chosen. This choice is however only an improvement if it makes the starting time and the ending time of the interval

```
void _Insert(AST_type *node,int start, int stop,int bandwidth,
              int iLevel) {
   int        ml, mr, i1;
   AST_type *pRunningNode;
   if (start==node->Interval.Start && stop==node->Interval.End)
      node->node_value = node->node_value + bandwidth;
   else {
      for (i1=0;i1<Split[iLevel];i1++) {
         pRunningNode = node->Children + i1;
         if ( start < pRunningNode->Inteval.End ) {
            if (pRunningNode->Interval.End < stop) {
               _Insert(pRunningNode, start,
                       pRunningNode->Interval.End, bandwidth,
                       iLevel+1);
               start = pRunningNode->Interval.End;
            } else {
               _Insert(pRunningNode,start,stop,bandwidth,
                        iLevel+1);
               break;
            }
         } /* if */
      } /* for */
      ml = node->max_value;
      for (;i1>=0;i1--) {
         mr = node->Children[i1].node_value+
              node->Children[i1].max_value;
         if ( mr>ml )
            ml=mr;
      }
      node->max_value=ml;
   }
}
void Insert(AST_type Data, reservation_type R) {
   _Insert(&Data,R.Interval.Start,R.Interval.End,R.BW,0);
}
```

Algorithm 3: Advance Tree Insertion in C

the same as the working hours.

An example of a tree divided with respect to humans are shown in Fig. 2.

3.2.2 Early rejection

The sooner we can detect that a reservation will not fit, due to the fact that there is not enough bandwidth left to reserve, the better. In the description of the `Insert` function we assumed that the check is done independently of the insertion. In this section we give some hints how to incorporate the check into the `Insert` function

```
void Delete(AST_type Data, reservation_type R) {
   _Insert(&Data,R.Interval.Start,R.Interval.End,0-R.BW,0,0);
}
```

Algorithm 4: Advance Tree Delete in C

Inclusion of check within insertion

In order not to reserve more bandwidth than can be delivered the `Insert` function has to check for over-reservations. One way to do this is when the `Insert` function discovers that there will be an over-reservation, the recursion stops, and undo of the insertions in the nodes so far is performed. Another way is that the recursion stops, marks the current node, and then a clean up function is called with the same arguments as the `Insert` function before. The clean up function can be implemented as a `Delete` function that only deletes the values up to the marked node and then stops. The second approach gives simpler code, fewer boolean checks, and therefore is also somewhat faster.

Order of traversal

The probability that a new reservation will not fit within the reservable bandwidth is somehow bigger at larger intervals. Hence during the recursion, if an interval has to be divided into smaller parts, the recursion first follows the largest interval of the edge parts of the interval, that is max $\left(I_{Q_1}, I_{Q_{rightmost}}\right)$ (see Fig. 5).

3.2.3 Locality

Since our tree is spanning a large time interval, we can assume that there will be some form of locality in the reservations and queries made. The locality gives additional improvements due to the fact that parts of generated data used during previous operations is already in the cache (cf. cache-aware data structures [1–3]) For our data structure we have developed a method of start the traversal of the tree inside the tree and not always from the top. In order to do that we need to find the lowest node that spans both the last interval, I_{Last} and the current interval, $I_{Current}$, that is the lowest node that spans the interval $I_{Merged} = [\min\left(S_{Last}, S_{Current}\right), \max\left(E_{Last}, E_{Current}\right)]$. This can be achieved by using the set X and a table of $|M_l|$ (eq. (1)). The operations on our data structure collect information from the top node and down and we need to maintain this information even when starting from the inside of the tree. This is done by using a stack in which all accumulated nv's are stored during the recursion down to the uppermost node in which I_{Last} was divided. When, for instance, querying the interval $I_{Current}$, we do a binary search among the $(L = O\left(\log n\right))$ levels in the stack to find the level, l, that has a node, N, spaning I_{Merged}. Then the recursion starts in N with the start-nv from level l in the stack. The running time of the search for N is $O(\log\log n)$.

Similar results

There is a resemblance between the searching in this data structure and a repeated search in a B-tree. Guiba et al. ([7]) have proposed an algorithm to traverse a B-tree without always having to start from the top node. They are using a lot of fingers (pointers) between nodes in the tree and a special search pattern. They use the search

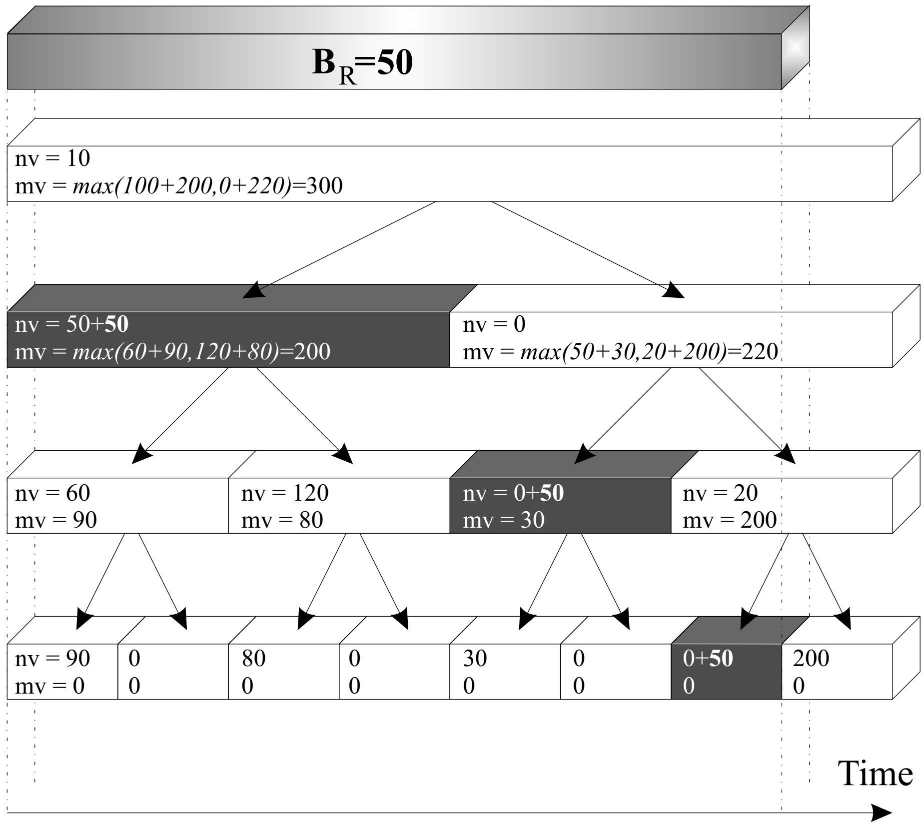

Figure 6: An example showing an insertion in the data structure.

pattern to find the lowest common ancestor. If the node to find is to the right of the last node the pattern will be:

- “go to right neighbour”, “go to father” and so on.

If the next node is to the left of the last node the pattern is:

- “go to left neighbour”, “go to father” and so on.

Since in our operations the node value needs to be accumulated from the root and down on the search path, Guiba et. al.’s proposal can not be directly used in our case. Further, the idea to perform a binary search on the complete path from the root to a given node is not new and was already employed in [6].

3.2.4 Time goes

In our modified segment tree the nodes are not moved into new positions and new nodes are not inserted, neither are nodes in the tree deleted. Hence, the time frame of the tree is static. But as time moves ahead towards the future, more reservations will be inserted more and more to the right parts of the tree. Eventually the intervals of the reservations will be so far off into the future that the interval will miss the data structure. A solution is to make the data structure wrap the spanning interval. Let the spanning interval of the data structure be twice as large as is believed to be needed. Then when the current time passes the first half of the entire interval, the interval is wrapped so that the first half becomes the third, and so on.

4 Conclusions

In this paper we presented a discrete data structure for reservations of limited resources over a time-span. The data structure is made with bandwidth reservations on the Internet in mind, but it is quite generic. It can be applied whenever a limited resource shall be reserved. The data structure proposed has time complexity independent of the number of reservations made and the size of the interval. We defined three operations on the data structure; `Insert` to insert a reservation, `Delete` to delete a reservation, and `MaxReserved` to query how much bandwidth is used during a query interval. The worst case time complexity of all operations is $O(\log n)$. In the second part of the paper we simplified the data structure by making it implicit. We conclude the paper with a number of suggestions how to improve the solution.

References

[1] P. Agarwal, L. Arge, A. Danner, and B. Holland-Minkley. On cache-oblivious multidimensional range searching. In *Proceedings of the 19th ACM Symposium on Computational Geometry*. ACM Press, 2003.

[2] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proceedings of the 34th ACM Symposium on Theory of Computing*, pages 268–276. ACM Press, 2002.

[3] L. Arge, L. Toma, and J. S. Vitter. I/O-efficient algorithms for problems on grid-based terrains. In *Proceedings of the 2nd Workshop on Algorithm Engineering and Experimentation (ALENEX '00)*, 2000.

[4] A. Brodnik. *Searching in Constant Time and Minimum Space (*MINIMÆRES MAGNI MOMENTI SUNT*)*. PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, 1995. (Also published as technical report CS-95-41.).

[5] M. L. Fredman. The complexity of maintaining an array and computing its partial sums. *Journal of the ACM*, 29(1):250–260, January 1982.

[6] Gaston H. Gonnet and J. Ian Munro. Heaps on heaps. *SIAM Journal on Computing*, 15(4):964–971, November 1986.

[7] L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts. A new representation for linear lists. In *Conference Record of the Ninth Annual ACM Symposium on Theory of Computing*, pages 49–60, Boulder, Colorado, 2–4 May 1977.

[8] T. Husfeldt and T. Rauhe. Hardness results for dynamic problems by extensions of fredman and saks' chronogram method. In *Proc. 25th Int. Coll. Automata, Languages, and Programming*, number 1443 in Lecture Notes in Computer Science, pages 67–78. Springer-Verlag, 1998.

[9] K. Mehlhorn. *Data structures and algorithms 3: Multi-dimensional searching and computational geometry*. Springer-Verlag, 1984. 91-032.

[10] O. Schelén, A. Nilsson, J. Norrgard, and S. Pink. Performance of QoS agents for provisioning network resources, 1999.