

Data Structure for a Time-Based Bandwidth Reservations Problem

Andrej Brodnik*[†]

Andreas Nilsson[†]

February 6, 2003

Abstract

We discuss a problem of handling resource reservations. The resource can be reserved for some time, it can be freed or it can be queried what is the largest amount of reserved resource during a time interval. We show that the problem has a lower bound of $\Omega(\log n)$ per operation on average and we give a matching upper bound algorithm. Our solution also solves a dynamic version of the related problems of a prefix sum and a partial sum.

1 Introduction

In Computer Communications we need to make bandwidth reservations over the Internet to provide *Quality of Service (QoS)* for the end users. The IETF (Internet Engineering Task Force) defined a standard for Integrated Services in routers ([12, 25]) and the end-to-end reservation setup protocol RSVP ([7]). Since the protocol does not scale well ([16]) IETF came up with a new approach, known as *differentiated services (diffserv)*, [5]). Schelén et. al. ([20, 21]) used the *diffserv* to design a new QoS architecture. In this architecture they provide virtual leased lines using the differentiated services to perform admission control through the system of agents. The agents work on per-hop basis and they need to maintain a database of the reservations made on their hop. In the backbone of the Internet it will most likely be many reservations to administrate and hence the use of an efficient data structure will be required. Moreover, in the design of the agents the authors propose that a single agent administrates several hops to make it more attractive for the ISP's (Internet Service Provider). Such a scenario even increases the need for use of an efficient data structure. Therefore Schelén et. al. in [19] proposed a solution that was, however, limited to a predefined set of possible time intervals over which the reservation could be made.

The bandwidth reservation problem is a special case of a more generic problem, where we need to administrate a limited resource over the time; e.g. use of human resources, computational power of super-computer, pool of cars etc. Although the solution in this paper covers all these problems, we use the term *bandwidth* when we talk about the reserved resource.

Definition 1 *In the bandwidth reservation problem we have a fixed amount of bandwidth to administer. Customers want to make reservations $R = \{B, I\}$ for a part of the bandwidth B during a time interval $I = [t_0, t_1)$ ($t_0 < t_1$, the interval starts at time t_0 and ends at time t_1 , and it includes t_0). The operations to support, besides initialization and destruction, are:*

- **Reserve**(B, I), that reserves B units of bandwidth for the time period $I = [t_0, t_1)$, where $t_0 \leq t_1$.
- **Free**(B, I), that frees the reserved bandwidth B during the interval I . Note that freeing the bandwidth is the same as making a reservation with a negative bandwidth.
- **MaxReserved**(I), that returns maximum reserved bandwidth during the interval I .

For the sake of clarity, we sometimes use the subscripts q and r for queries and reservations respectively. For example, a reservation interval $I_r = [t_{r0}, t_{r1})$. In the paper we also use the notation $\max(x, y)$ denoting a function returning the bigger of x and y .

* 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

1.1 Literature background

In the literature we could not find any reference to the bandwidth reservation problem with an arbitrary reservation interval – i.e. interval where endpoints are not drawn from a predefined set. However, the problem is similar to problems we find in other fields of computer science that handle intervals on a real line (e.g. computational geometry, dynamic computation and geometric search [3, 9, 10, 18]). These problems are generally solved using *segment trees* ([17, 22]), which were introduced by Bentley ([4]) as a solution to the Klee’s rectangle problem ([14]). The limitation in all these problems is that the end-points of the intervals belong to a *fixed set* of points. In our problem we have no such a set.

Kuchem et. al. in ([15]) presented in a way similar data structure to ours, although it still deals with a fixed set of points. They use the structure in a VLSI design. Bose et. al. independently developed in [6] a similar data structure to solve a number of geometric problems.

Another pair of related problems are the well studied partial sum problem ([11], brief in [13]), and the prefix sum problem ([11]). In the prefix sum problem we have an array $V(i), 1 \leq i \leq n$ on which we want to perform these two operations: (1) `Update(i, x): $V(i) = V(i) + x$` ; and (2) `Retrieve(m): $\sum_{k=1}^m V(k)$` for arbitrary values of i, x and m . In [11] Fredman shows a lower bound of $\Omega(\log n)$ for the problem under the comparison based model. In the same paper Fredman also presents an algorithm with a matching upper bound.

In the rest of the paper we first show that the logarithmic lower bound carries over to the bandwidth reservation problem. We continue with a presentation of a data structure we call *BinSeT* (binary segment tree) that gives us a matching upper bound. We conclude the paper with final remarks.

2 Lower bound

Theorem 1 *Given an arbitrary sequence of operations from a bandwidth reservation problem, each of them requires at least $\Omega(\log n)$ comparisons on the average, where n is the number of intervals we are dealing with.*

Proof: Assume that we have a solution to the bandwidth reservation problem that requires $o(\lg n)$ time. We will show how to use such a solution to solve the prefix sum problem in time $o(\log n)$ which contradicts the lower bound by Fredman ([11]).

First, we introduce an extra point $n + 1$ right to all other points representing $+\infty$. It is needed since in our problem we are dealing with open intervals on the right side. Next, we translate the array of elements in the prefix sum problem into end-points of intervals. More precisely, the $V(i)$ element of the array is represented by the interval that starts at point i and ends at the right most point $n + 1$: $[i, n + 1)$. Therefore, the reserved bandwidth at point p is the sum of all reserved bandwidths for intervals starting at points j , where $1 \leq j \leq p$. This gives us the following translation of prefix-sum problem operations:

- the operation `Update(i, x)` into `Reserve($x, [i, n + 1)$)`; and
- the operation `Retrieve(j)` into a query `MaxReserved($[j, j + 1)$)`.

This translation gives us an $o(\lg n)$ solution to the prefix sum problem and hence contradicts the lower bound by Fredman. *QED*

Note that, the prefix sum as presented by Fredman ([11]) is also a *static problem* – i.e. the array of elements neither expands nor shrinks. On the other hand, the solution we present in the following section does support insertion of new points (intervals) and deletion of points (intervals). Hence, by using the translation in the proof we also get a logarithmic solution to the dynamic version of the prefix-sum problem.

3 Upper bound

To prove an upper bound we use a data structure called *BinSeT* that supports the required operations in logarithmic time. Before going into details of data structure we describe how we represent reservations.

3.1 Representation of reservations

We do not represent a reservation interval as a single entity, but we split it into two, what we call, *reservation events*. A reservation event is a point in time when an increase or decrease in the amount of a reserved bandwidth occurs. For example, we store a reservation $R = \{B, [t_0, t_1]\}$ as reservation events $E_0 = (t_0, +B)$ and $E_1 = (t_1, -B)$. In other terms, we convert an interval $[t_0, t_1]$ into two semi-infinite intervals $[t_0, +\infty)$ and $[t_1, -\infty)$. Hence, the operations from Definition 1 are converted:

- **Reserve** $(B, [t_0, t_1])$ into adding of reservation events $E_0 = (t_0, +B)$ and $E_1 = (t_1, -B)$; and
- **Free** $(B, [t_0, t_1])$ into adding of reservation events $E_0 = (t_0, -B)$ and $E_1 = (t_1, +B)$; while
- **MaxReserved** $([t_0, t_1])$, remains the same.

If we want to store extra information with each reservation we introduce an additional dictionary data structure to store this information and bind the reservation events to records in the dictionary.

3.2 Data Structure

The binary segment tree BinSeT is a data structure that combines properties of a binary and a segment tree. The former permits dynamic insertion and deletion of reservation events and the later answering queries about the maximum reserved bandwidth. In detail, the leaves represent and store information about the reservation events, while each internal node covers a segment (interval) $I = [t_0, t_1]$ and stores information about the values (bandwidth) on that interval. To ensure $O(\log n)$ worst case performance, we balance BinSeT tree as an AVL tree (cf. [1, 8, 24]) – hence we also need to talk about the height of BinSeT tree. This gives the following invariance for every node of our data structure:

Invariance 1 *The information stored with the node n representing an interval $I = [t_0, t_1]$ is the maximum value μ_n on the interval and the change δ_n of the value on the interval. Besides, with a node is also stored the left-most event in the right subtree $t_0 < \tau < t_1$. The difference of heights of left and right subtree is at most one.*

Note, if a node covers interval $[t_0, t_1]$, the left subtree covers interval $[t_0, \tau)$ and the right subtree the interval $[\tau, t_1)$.

In simpler terms, in the BinSeT tree each node has its own *local* system of reserved resource values on its interval. The system is offset to the global so, that in the beginning of the interval the value is considered to be 0. To get total (global) value of reserved resources one has to add δ -s for all left siblings on the path from the node to the root.

It is easy to verify the following lemma:

Lemma 1 *Let l be left child and r right child of an internal node n . Then the equations:*

$$\begin{aligned} \delta_n &= \delta_l + \delta_r \\ \mu_n &= \max(\mu_l, \delta_l + \mu_r) \end{aligned} \tag{1}$$

hold for all nodes n .

The detail data structure is represented in Algorithm 1. The structure is slightly different from the one described above since it does not include times t_0 and t_1 , but only the τ . However, values t_0 and t_1 can be implicitly calculated during recursive descend. At this point we note two things: first, a node has either two sub-trees (an internal node) or none (a leaf); and second, a leaf stores in both δ and μ the amount of the reserved bandwidth at the reservation event it represents, and in τ the time of the event. As a consequence of the first observation we conclude, that the number of internal nodes is one less than the number of leaves. Since the number of leaves is at most $2n$, where n is a number of reservation intervals, this proves the following lemma, under the RAM model:

Lemma 2 *The size of the BinSeT storing n reservation events is $\Theta(n)$ words.*

```

typedef struct _sBinSeT {
    tResource       $\mu$ ;
    tResource       $\delta$ ;
    tTime           $\tau$ ;
    unsigned int   height;
    struct _sBinSeT* left;
    struct _sBinSeT* right;
} tBinSeT;

```

Algorithm 1: Binary segment tree definition.

3.3 Operations

Finally we describe how to implement efficiently queries and adding of reservation events. All our solutions will be recursive and will start traversing the data structure from the root. We assume that we store with BinSeT also the time of the first (t_f) and the last (t_l) reservation event. These are also times t_0 and t_1 , respectively, for the root of the complete BinSeT. If we descend in the left subtree, then the t_0 and t_1 for this subtree become values t_0 and τ , respectively, of the current root. We treat similarly the right subtree. This is also the reason why we need not store values t_0 and t_1 with a node.

We start with a query MaxReserved (see Algorithm 2). Assuming Invariance 1 we prove:

```

tResource MaxReserved(tBinSet* node, tTime t0, tTime t1, tInterval query) {
    tResource leftMax, rightMax;
    tInterval queryAux;
    if ((t0 == query.t0) && (t1 == query.t1))          /* whole interval – stopping condition */
        return node-> $\mu$ ;
    if (query.t1 <= node-> $\tau$ )                             /* query in left subinterval */
        return MaxReserved(node->left, t0, node-> $\tau$ , query);
    if (node-> $\tau$  <= query.t0)                             /* query in right subinterval */
        return node->left-> $\delta$  +
            MaxReserved(node->right, node-> $\tau$ , t1, query);
    queryAux= query; queryAux.t1= node-> $\tau$ ;             /* query in both subinterval – so split it */
    leftMax= MaxReserved(node->left, t0, node-> $\tau$ , queryAux);
    queryAux= query; queryAux.t0= node-> $\tau$ ;
    rightMax= MaxReserved(node->right, node-> $\tau$ , t1, queryAux);
    return max(leftMax, node->left-> $\delta$  + rightMax);
} /* MaxReserved */

```

Algorithm 2: Query MaxReserved in BinSeT.

Lemma 3 *The query MaxReserved in BinSeT takes $O(\log n)$ worst case time.*

Proof: The correctness of the proof uses induction. Due to the limited presentation space we give only a justification of the induction step. Let the query be for the interval $I_q = [t_{q0}, t_{q1})$ and let the node cover interval $[t_0, t_1)$. Then we have the following possibilities:

- If $t_{q0} = t_0$ and $t_{q1} = t_1$, the answer is exactly μ of the node.
- If $t_{q1} \leq \tau$ then the answer is the same as the answer to the same query I_q in the left subtree `left` covering the interval $[t_0, \tau)$.

Similarly, if $\tau \leq t_{q0}$ then the answer is the same as the answer to the query I_q in the right subtree `right` covering the interval $[\tau, t_1)$. However, due to Lemma 1 we have to add left node's δ .

- Finally, in the most general case when $t_0 < t_{q0} < \tau < t_{q1} < t_1$ the answer is because of Lemma 1

$$\max(\text{MaxReserved}(\text{left}, [t_{q0}, \tau)), \text{left} \rightarrow \delta + \text{MaxReserved}(\text{right}, [\tau, t_{q1}])) .$$

To see that the running time of the query is logarithmic, i.e. proportional to the height of the BinSeT, observe that the third case occurs only once, while the tree is balanced in the AVL-sense. *QED*

The last operation is Add that adds a reservation event. Note, that we *never* explicitly delete a reservation event, we might just add a reservation event with a negative value (see section 3.1).

Lemma 4 *Adding of a reservation event into BinSeT can be done in $O(\log n)$ worst case time.*

Proof: Let us assume that we are adding a reservation event at time t_r and for the value B_r . We start (see

```

tBinSet* Add(tBinSet* node, tTime  $t_r$ , tResource  $B_r$ ) {
  if (node->left != null) { /* WE ARE NOT AT THE LEAF YET. */
    if ( $t_r < node->\tau$ ) {
      node->left = Add(node->left,  $t_r$ ,  $B_r$ );
      if (node->left == null) { /* we lost the leaf */
        free(node); return node->right; /* but we need no rebalancing */
      }
    } else { ... } /* similarly for the right subtree */
    node-> $\delta$  +=  $B_r$ ; /* update  $\delta$  and  $\mu$  - see eq. (1) */
    node-> $\mu$  = max(node->left-> $\mu$ , node->left-> $\delta$  + node->right-> $\mu$ );
    node = Rebalance(node);
    return node;
  } /* WE ARE AT THE LEAF. */

  if ( $t_r \neq node->\tau$ ) return Insert(node,  $t_r$ ,  $B_r$ );
  else {
    node-> $\mu$  = node-> $\delta$  = node-> $\mu$  +  $B_r$ ;
    if (node-> $\mu$  != 0) return node;
    else { free(node); return null; }
  }
} /* Add */

```

Algorithm 3: Adding of a reservation event in BinSeT.

Algorithm 3) at the root and recursively descend to the leaves. The decision into which subtree to descend is based on the node's value τ and t_r : when $t_r < \tau$, we descend into the left subtree and otherwise into the right one. Note, we always go all the way to the leaves.

The time τ of the reached leaf can be either the same as t_r or not. If it is not, we create a new internal node `newNode` and make the reached leaf one of its leaves. Besides, we create a new leaf with an added reservation event and properly update the values. For details see Algorithm 4.

On the other hand, if $\tau = t_r$ we add value B_r to leaf's values μ and δ . If new values are not 0 we are done. However, if they are 0 we have to delete the leaf and replace its parent with leaf's sibling. We also delete the parent. On the way back to the root we update δ -s and μ -s as required in eq. (1). Algorithm 3 gives a skeleton of the algorithm.

It remains to describe the rebalancing of BinSeT (see call of `Rebalance` function in Algorithm 3). Since BinSeT is an AVL-like tree, we rebalance it using regular single and double rotations. While the details of when and how to perform the rotations are explained in most textbooks (cf. [8, 24]) we concentrate only on updates of values μ and δ . Observe that the value τ does not change during rotations.

First consider a single rotation shown in Figure 1 (we are omitting description of a mirroring single rotation). The new values of nodes b and d , they are marked with a prime sign, are computed using the formulae:

$$\begin{aligned}
 d.\delta' &= b.\delta & d.\mu' &= b.\mu \\
 b.\delta' &= b.\delta - E.\delta & b.\mu' &= \max(A.\mu, A.\delta + C.\mu) \quad \text{by eq. (1)}
 \end{aligned}
 \tag{2}$$

Observe, that the order in which new values are computed is important: therefore we first compute δ and μ values at d and afterwards at b .

Similarly we compute new values in double rotation (cf. Figure 2):

```

tBinSet* Insert(tBinSet* oldLeaf, tTime  $t_r$ , tResource  $B_r$ ) {
    tBinSet* newLeaf;
    tBinSet* newNode;

    /* First make a new leaf out of an inserted event: */
    newLeaf= (tBinSet*) malloc( sizeof(tBinSet) );
    newLeaf-> $\mu$ = newLeaf-> $\delta$ =  $B_r$ ; /* set first as a segment tree */
    newLeaf-> $\tau$ =  $t_r$ ;
    newLeaf->height= 1; /* and then as a binary tree. */
    newLeaf->left= newLeaf->right= null;

    /* And then make a new internal node: */
    newNode= (tBinSet*) malloc( sizeof(tBinSet) );
    newLeaf->height= 2; /* now first set as a binary tree */
    if (oldLeaf-> $\tau$  <  $t_r$ ) { newNode->left= oldLeaf; newLeaf->right= newLeaf; }
    else { newNode->left= newLeaf; newLeaf->right= oldLeaf; }
    newNode-> $\delta$ = newNode->left-> $\delta$  + newNode->right-> $\delta$ ; /* and then as a segment tree */
    newNode-> $\mu$ = max(newNode->left-> $\mu$ ,
                    newNode->left-> $\delta$  + newNode->right-> $\mu$ );
    return newNode;
} /* Insert */

```

Algorithm 4: Insertion of a new reservation event in BinSeT.

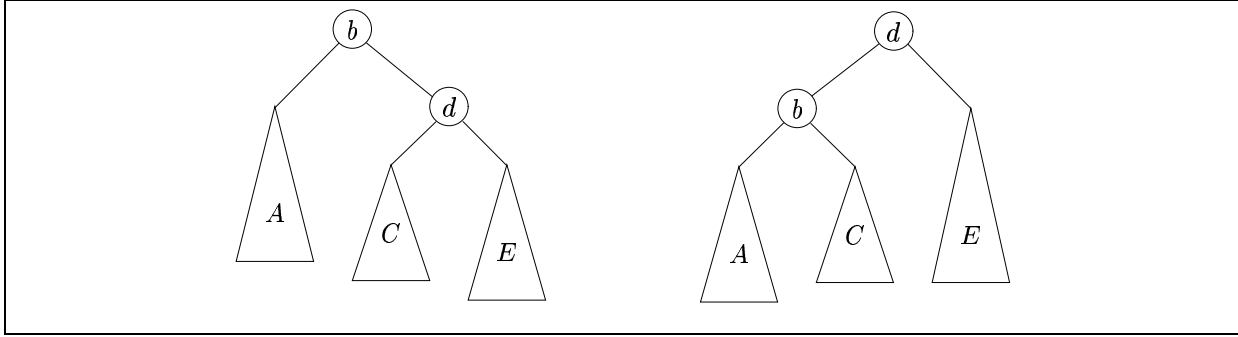


Figure 1: Single rotation.

$$\begin{aligned}
d.\delta' &= b.\delta & d.\mu' &= b.\mu \\
f.\delta' &= f.\delta - C.\delta & b.\mu' &= \max(A.\mu, A.\delta + C.\mu) \text{ by eq. (1)} \\
b.\delta' &= f.\delta' & f.\mu' &= \max(E.\mu, E.\delta + G.\mu) \text{ by eq. (1)}
\end{aligned} \tag{3}$$

To prove the correctness of Algorithm 3 we need to see that it preserves Invariance 1. First, if a new reservation point is added in the interval the δ should be changed exactly for this value. This is done in line 9 of Algorithm 3. In the following line new μ is computed according to eq. (1) and hence also this part of invariance is kept.

Finally, the rebalancing keeps the difference in heights between the left and right subtrees always at most one. Consequently, the height of the tree is $O(\log n)$ and the running time of Algorithm 3 is also $O(\log n)$. \mathcal{QED}

Our data structure uses AVL-like balancing technique, but it could use any one. For more details on balancing and balance binary trees see [2, 23] or any other text book.

This brings us to the final theorem:

Theorem 2 *The Bandwidth Reservation Problem can be solved under the comparison based machine model in $\Theta(\log n)$ time per operation and in $\Theta(n)$ words of space. This is tight.*

Obviously it is straight forward to adapt the solution to handle also queries of the minimum reserved bandwidth. Moreover, using the translation in Theorem 1 we also get a logarithmic time solution to the dynamic versions of partial sum problem and of prefix sum problem.

A practical improvement is to store with a node not its δ and μ , but rather its children's μ -s and left child's δ (the right child's δ is actually never used!). Using this information it is easy to compute also node's μ using eq. (1). One would think that the size of the data structure increases after such a modification. But it does not, since we do not

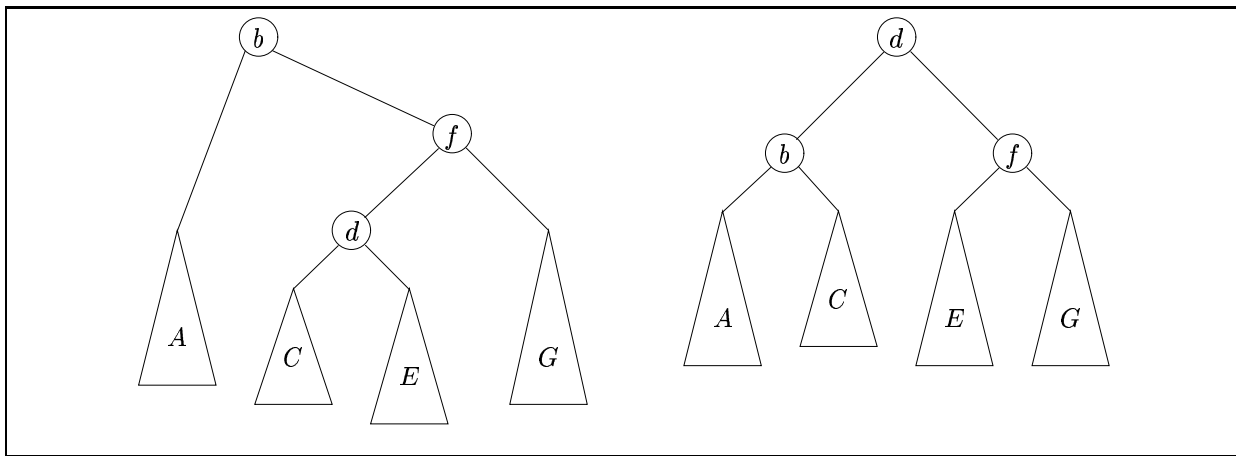


Figure 2: Double rotation.

need leaves at all. Moreover, since in Algorithm 3, and in eq. (2) and eq. (3) we no more need to access children, everything runs faster because of fewer cache misses.

4 Conclusions

We showed that the data structure BinSeT (binary segment tree) solves the dynamic version of the Bandwidth Reservation Problem optimally (space- and time-wise) under the comparison based model. The solution requires $\Theta(\log n)$ time for the queries and updates and $\Theta(n)$ space. It substantially improves solution presented in [19] which restricted the maximum allowed reservation intervals and their smallest granularity.

Using BinSeT we also solve dynamic versions of prefix sum and partial sum problems. Interesting enough, asymptotically the dynamic solution has the same time and space complexity as the static version.

There are a number of open problems left. For example, what are lower and upper bounds under the cell probe model and bounded universe? Interesting question is also whether can we benefit from the fact that time always increases? At least on the average?

References

- [1] G. M. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. In *Soviet Math. Doklady* 3, pages 1259–1263, 1962.
- [2] A. Andresson. *Efficient Search Trees*. Ph. D. Thesis, Department of Computer Science, Lund University, Sweden, 1990.
- [3] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. Technical Report RS-96-28, BRICS, Univ. of Åarhus, Denmark, 1996.
- [4] J.L. Bentley. Algorithms for Klee’s rectangle problems. Computer Science Department, Carnegie-Mellon University, Pittsburgh, 1972.
- [5] S. Blake et al. An architecture for differentiated services. RFC (Informational) 2475, IETF, December 1998.
- [6] P. Bose, M. van Kreveld, A. Maheshwari, P. Morin, and J. Morrison. Translating a regular grid over a point set. *Computational Geometry: Theory and Applications*. Accepted for publication.
- [7] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource reservation protocol (RSVP) – version 1 functional specification. Request for Comments (Proposed Standard) 2205, Internet Engineering Task Force, September 1997.

- [8] F. M. Carrano and J. J. Prichard. *Data Abstraction and Problem Solving using JAVA (Walls and Mirrors)*. Addison-Wesley Pub Co, 2001.
- [9] A. Chan, F. Dehne, and A. Rau-Chaplin. Coarse-grained parallel geometric search. *Journal of Parallel and Distributed Computing*, 57(2):224–235, 1999.
- [10] D. Eppstein. Dynamic three-dimensional linear programming. In *IEEE Symposium on Foundations of Computer Science*, pages 488–494, 1991.
- [11] M.L. Fredman. The complexity of maintaining an array and computing its partial sums. *Journal of the ACM*, 29(1):250–260, January 1982.
- [12] R. Guerin, C. Partridge, and S. Shenker. Specification of guaranteed quality of service. Request for Comments (Proposed Standard) 2212, Internet Engineering Task Force, October 1997.
- [13] 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.
- [14] V. Klee. Can the measure of $\bigcup_1^n [a_i, b_i]$ be computed in less than $O(n \log n)$ steps? *Amer. Math. Monthly*, 84:284–285, 1977.
- [15] R. Kuchem, D. Wagner, and F. Wagner. Optimizing area for three-layer knock-knee channel routing. *Algorithmica*, 15(5):495–519, May 1996.
- [16] A. Mankin, F. Baker, B. Braden, S. Bradner, M. O’Dell, A. Romanow, A. Weinrib, and L. Zhang. Resource reservation protocol (RSVP) – version 1 applicability statement and some guidelines on deployment. RFC (Informational) 2208, IETF, September 1997.
- [17] K. Mehlhorn. *Data structures and algorithms 3: Multi-dimensional searching and computational geometry*. Springer-Verlag, 1984. 91-032.
- [18] K. Mehlhorn and F.P. Preparata. Routing through a rectangle. *Journal of the ACM*, 33(1):60–85, 1986.
- [19] O. Schelen, A. Nilsson, J. Norrgard, and S. Pink. Performance of QoS agents for provisioning network resources. In *IEEE/IFIP IWQoS 99*, 1999.
- [20] O. Schelén and S. Pink. An agent-based architecture for advance reservations. In *IEEE 22nd Annual Conference on Computer Networks (LCN’97)*, Minneapolis, Minnesota, November 1997.
- [21] O. Schelén and S. Pink. Sharing resources through advance reservation agents. In *Proceedings of IFIP Fifth International Workshop on Quality of Service (IWQoS’97)*, New York, May 1997.
- [22] M.I. Shamos and F.P. Preparata. *Computational geometry*. Springer, 1985.
- [23] T.H. Cormen and C.E. Leisserson and R.L. Rivest and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, McGraww-Hill Press, 2001.
- [24] M.A. Weiss. *Data Structures and Problem Solving Using Java*. Addison-Wesley Pub Co, 1997.
- [25] J. Wroclawski. Specification of the controlled-load network element service. Request for Comments (Proposed Standard) 2211, Internet Engineering Task Force, October 1997.

A An Example

Bottom of Figure 3 gives an example of a reservations made during 16 time slots. In the upper part of the figure is presented a BinSeT tree as build over the presented reservations. Additional arrows explain how particular values of δ , μ and τ are computed from the reservations.

In Figure 4 is shown a detail from the example. It presents “local systems” mentioned in § 3.2 for internal nodes *A* and *C*. The systems are presented with two different patterns: the first is expanding over slots 8 to 12 and the second one from 12 to 16. The figure also depicts δ and μ values for both nodes.

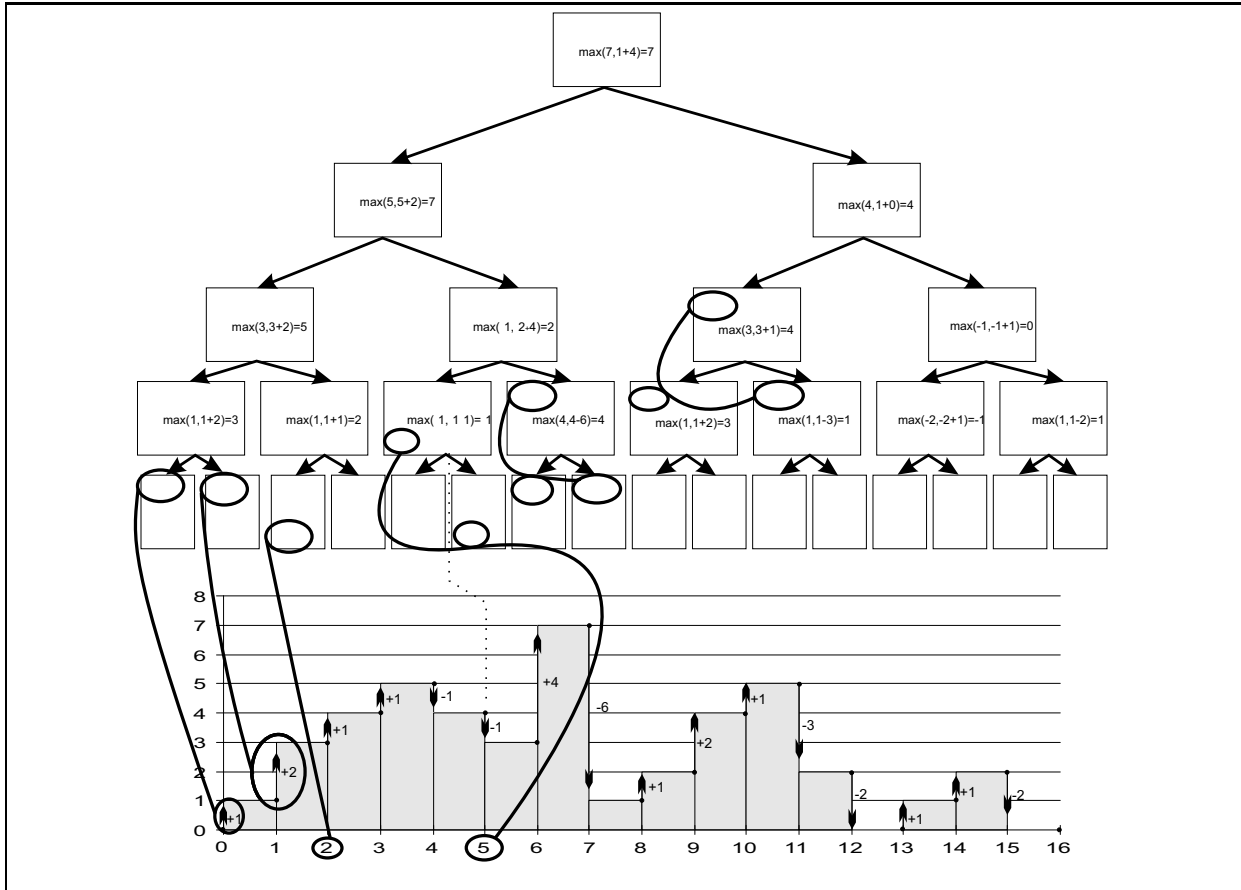


Figure 3: Example of a BinSeT tree.

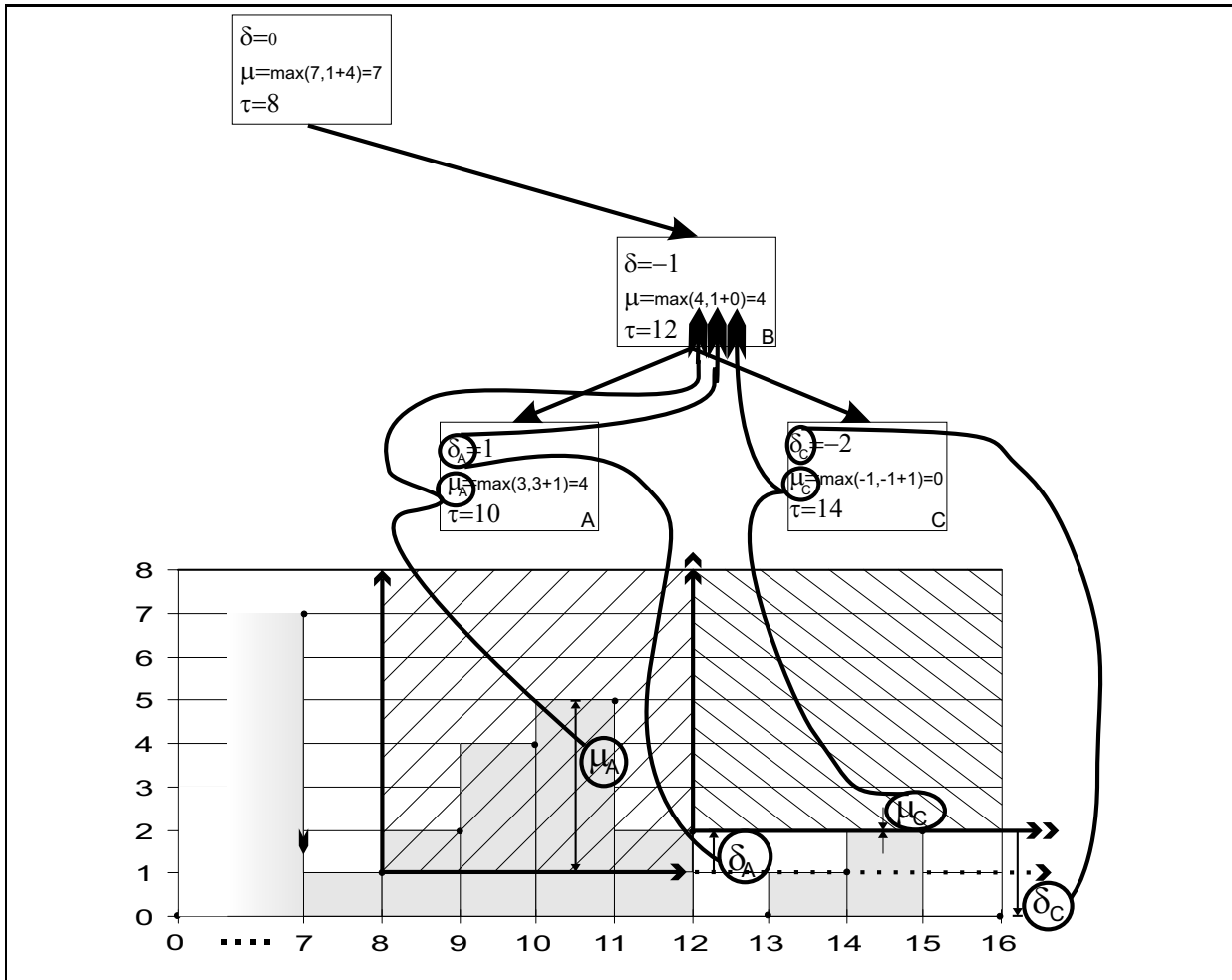


Figure 4: Local systems of internal nodes A and C.